Table des matières

Déclaration i
Remerciements ii
Résumé iii
Liste des figures vi
1. Introduction 1
2. Réseaux de neurones 2
2.1 Principe et origine 2
2.2 Le Perceptron multi couches 4
2.3 Les variantes actuelles 7
2.3.1 CNN 7
2.3.1.1 Origine et utilisation actuelle 7
2.3.2 RNN 10
2.3.2.1 Origine et utilisation actuelle 10
2.3.2.2 Vanishing / Exploding gradient 14
2.4 Lequel choisir ? 15
3. Long Short-Term Memory (LSTM) 17
3.1 Fonctionnement 17
3.2 Particularités et paramètres 23
3.2.1 Word embeddings 24
3.2.1.1 Encodage One-hot 25
3.2.1.2 TF-IDF (Term Frequency-Inverse document Frequency) 26
3.2.1.3 Word2Vec 26
3.2.1.4 GloVe (Global Vectors for Word Representation) 28
3.2.1.5 FastText 30
3.2.1.6 Poincaré embeddings 30
3.2.1.7 Choix 31
3.2.1.7.1 TF-IDF vs Word2Vec 32
3.2.1.7.2 Word2Vec vs GloVe 32
3.2.1.7.3 GloVe vs FastText 34
3.2.1.7.4 GloVe vs Poincaré 36
4. Siamese networks 37
4.1 Fonctionnement 37
5. De la théorie à la pratique 40
5.1 TensorFlow 40
5.1.1 Les bases de Tensorflow 42
5.1.1.1 Tensor 42
5.1.1.2 Session 42
5.1.1.3 Variables 43
5.1.2 Graphes computationnels 44
5.1.2.1 Scope 46
5.2 Phase d'apprentissage 48

5.2.1 Bonnes pratiques 49
5.2.1.1 Général 49
5.2.1.2 Hyperparamètres 50
5.3 Phase de test 52
5.3.1 Bonnes pratiques 52
6. Création du modèle 52
6.1 Jeu de données 52
6.2 API Embeddings 53
6.3 Entraînement du modèle 56
6.4 Résultats 65
7. Test du modèle construit 68
7.1 Résultats 73
8. Synthèse et perspectives 74
9. Conclusion 76
Bibliographie 77

Liste des figures

Figure 1 : Le Perceptron 2
Figure 2 : Un neurone humain 2
Figure 3 Fonction linéaire 4
Figure 4 : Le Perceptron multi couches 5
Figure 5 : Descente du Gradient 6
Figure 6 : Image d'un « X » CNN 8
Figure 7 : Image d'un « X » décalé CNN 8
Figure 8 : Caractéristiques similaires CNN 9
Figure 9 Filtre CNN 9
Figure 10 : Schéma RNN 11
Figure 11 Fonctionnement technique d'un RNN 12
Figure 12 : Schéma RNN déplié 13
Figure 13 : Comparaison CNN et RNN 16
Figure 14 Simple RNN 17
Figure 15 Tangent hyperbolique Tanh 18
Figure 16 Architecture LSTM 19
Figure 17 Symboles schéma LSTM 19
Figure 18 Fonction Sigmoïde 20
Figure 19 Porte d'oubli LSTM 21
Figure 20 Porte d'entrée LSTM 22
Figure 21 Porte de sortie LSTM 23
Figure 22 Formule TF-IDF 26
Figure 23 Exemple utilisation Skip-gram 27
Figure 24 Matrice de co-occurrence 29
Figure 25 Exemple d'espace hyperbolique de la méthode Poincaré 31
Figure 26 Comparaison coût, RAM, temps de Word2Vec et GloVe 32
Figure 27 Comparaison précision GloVe vs Word2Vec 33
Figure 28 Word2Vec vs FastText sur le corpus « Brown » 35
Figure 29 Word2Vec vs FastText sur le corpus « text9 » 35
Figure 30 Siamese network 38
Figure 31 Cosinus entre deux vecteurs projetés 38
Figure 32 Angle de 30° entre deux vecteurs 40
Figure 33 TensorFlow API 41
Figure 34 Communauté de TensorFlow 41
Figure 35 Représentation d'un simple graphe computationnel 45
Figure 36 Représentation d'un graphe computationnel multi-GPU 45
Figure 37 Exemple de graphe avec TensorFlow 47
Figure 38 Fonction de coût et Learning rate 51
Figure 39 Quora paires de questions 53
Figure 40 Graphique de la précision du modèle sur le jeu d'entraînement 65
Figure 41 Graphique du résultat de la fonction de coût sur le jeu d'entraînement 66
Figure 42 Graphique de la précision sur le jeu de validation 67
Figure 43 Graphique du résultat de la fonction de coût sur le jeu de validation 67

1. Introduction

L'idée initiale de ce travail de recherche était de fournir aux professeurs, que ce soit de hautes écoles ou non, un outil qui leur permettrait de pouvoir corriger automatiquement les examens de leurs étudiants. Pour cela, le prototype aurait besoin d'une liste des questions, d'une liste des réponses justes et des copies des étudiants.

Le but sous-jacent de ce travail est d'être capable de pouvoir analyser la sémantique d'une phrase et de pouvoir la comparer avec une autre pour déterminer si elles sont similaires. Ces deux phrases ne seraient évidemment pas composées exactement des mêmes mots mais seraient différentes. C'est là tout le défi apporté par ce sujet de travail de Bachelor.

Ce mémoire se découpera en plusieurs axes où nous aborderons, dans un premier temps, les réseaux de neurones, ce qu'ils sont et les types existants à l'heure actuelle.

Il s'agira dans un deuxième temps de choisir une architecture de réseau de neurones, de faire des recherches quant à son utilisation et de voir si d'autres concepts y sont essentiels.

Ensuite, nous aborderons quelques bonnes pratiques à suivre dans l'élaboration d'un réseau de neurones artificiels avant de passer à la partie de prototypage.

Enfin, nous construirons notre propre réseau de neurones et nous l'évaluerons avant de passer aux perspectives d'améliorations et à la conclusion.

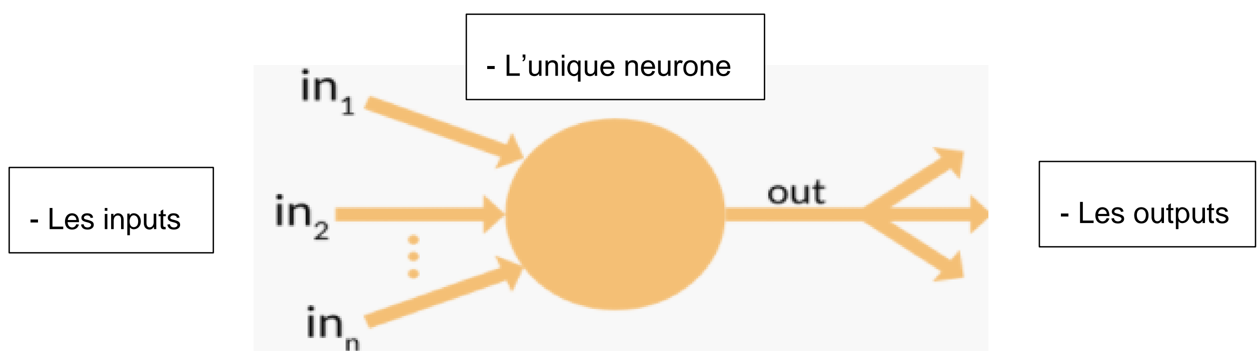
2. Réseaux de neurones

2.1 Principes et origine

Les réseaux de neurones dit « artificiels » sont des systèmes dont la conception est basée sur le fonctionnement du système neurologique d'un cortex cérébral, mais au niveau informatique et surtout à plus petite échelle. (Bheemaiah et al., 2017) Ceux-ci sont optimisés pour permettre des traitements probabilistes de manière performante et sont capables d'obtenir des résultats indépendamment des idées de son concepteur. (Wikipedia, 2018c) Ils sont très utilisés dans des problèmes de statistiques où il est par exemple nécessaire d'effectuer une classification automatique de consommateurs selon leur âge et leur sexe.

L'origine des réseaux de neurones artificiels a en fait commencé par la création du « Perceptron » qui a été inventé en 1958 par le psychologue américain Frank Rosenblatt.(Rosenblatt, 1958) Ce préambule ne parle pas d'un « réseau de neurones », mais il modélise plutôt le fonctionnement d'un seul et unique neurone avec des informations qui arrivent en entrée(inputs en anglais), un traitement en interne et une information en sortie(outputs en anglais). Voici comment on peut se représenter

Figure 1 : Le Perceptron

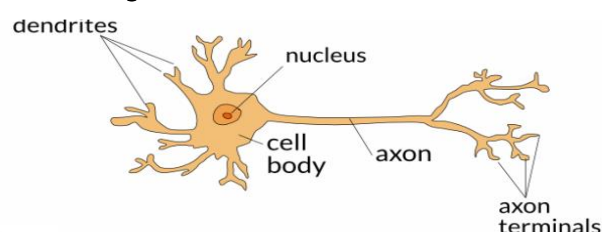


celui-ci :

(argodev, 2017)

Et voici comment un neurone est schématiquement représenté dans le fonctionnement du cerveau humain :

Figure 2 : Un neurone humain



(argodev, 2017)

On peut observer que le fonctionnement est le même, à savoir que le neurone humain reçoit ses informations sur des dendrites, qu'il effectue ce pour quoi il a été conçu dans le noyau et qu'il transmet le message au suivant via l'axone. Au même titre, le Perceptron reçoit des données en inputs, effectue ses traitements dans la couche qui contient un seul neurone et transmet ses résultats en outputs.

Dans la Figure 1, le premier niveau, représenté par les « in_1 », « in_2 » et « in_n », contient toutes les données en entrée que nous voulons traiter avec notre réseau pondéré par des poids. Par exemple, nous lui fournissons une liste de mails pour lesquels nous aimerions savoir ceux contenant des spams. Le deuxième niveau qui est représenté par « l'unique neurone » sur la Figure 1 va contenir une fonction d'activation des connexions neurologiques. Celle-ci va nous permettre d'ajuster les connexions en entrée du neurone en fonction du résultat obtenu avec cette fonction. Pour le dernier niveau, celui des outputs, il nous fournira un seul résultat qui sera la prédiction finale. Pour reprendre l'exemple des mails, l'output pourrait être le fait de savoir si un mail est bel et bien un spam ou non.

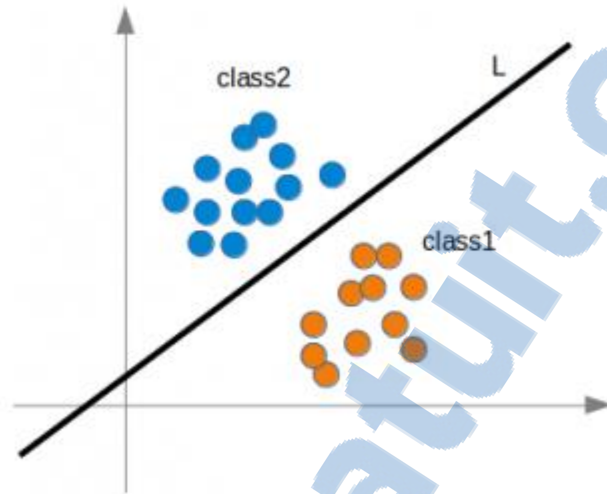
Dans son ensemble, le Perceptron fonctionne de la manière suivante :

Nous avons des entrées qui lui sont données avec un poids correspondant à chacune d'elles. (Un poids ou poids synaptique est un coefficient numérique qui est attribué à chacune des entrées d'un neurone, de manière aléatoire au début puis les poids sont adaptés au fur et à mesure, et qui permet de pondérer celles-ci.(Wikipedia, 2017) Le Perceptron va ensuite faire une somme pondérée (les entrées multipliées par son poids correspondant), il va ajouter un biais et pour finir, une fonction d'activation est appliquée. En fonction du problème à résoudre, la somme pondérée peut suffire, mais dans certains cas, une fonction d'activation peut être appliquée, tel que Sigmoide et Relu.(Statistica, 2013)

Un biais est utilisé dans les réseaux de neurones artificiels pour permettre de déplacer la tangente verticale pour qu'elle sépare plus efficacement les classes à prédire. La tangente verticale est une droite placée sur un espace à deux dimensions et permet de représenter de manière graphique la séparation entre deux classes, qui elles-mêmes sont deux états différents que peut prendre un attribut (homme ou femme, blond ou brun, riche ou pauvre…).

Si nous prenons le cas d'une fonction linéaire qui sépare deux classes de personnes, les hommes et les femmes, la représentation graphique se présentera ainsi :

Figure 3 : Fonction linéaire



(R, 2018)

Si nous fournissons les données d'une personne en entrée d'un réseau de neurones, on s'attend à ce qu'il puisse nous dire si cette personne est un homme ou une femme. Nous parlons ici d'un problème binaire pour que cela soit plus facilement compréhensible de manière graphique. Pour permettre de bien séparer les deux classes à prédire, nous utilisons un biais. Celui-ci est simplement une valeur que l'on rajoute au neurone et qui va permettre de déplacer la courbe pour qu'elle sépare bien les deux classes comme dans la Figure 3.(R, 2018, p. 20) Ce procédé est aussi utilisé lorsque l'on a besoin de faire une classification dans plus de deux classes données, sur le graphique nous verrions « n » groupes différents, n représentant le nombre de classes.

Une fois la fonction d'activation appliquée, le neurone compare son résultat avec le seuil d'activation et prédira l'une ou l'autre des sorties possibles.

Il est important de savoir que le Perceptron ne permet que la classification des inputs de manière binaire. C'est-à-dire que nous ne pouvons pas avoir plus de deux résultats finaux possibles et que seuls les problèmes linéaires peuvent être résolus.

2.2 Le Perceptron multi couches

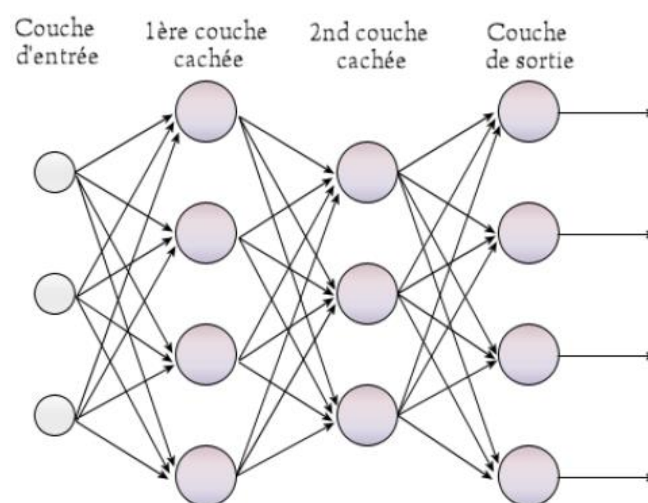
Le Perceptron qui ne contenait qu'un seul neurone a évolué pour permettre le traitement de problèmes non linéaires. Ainsi, il est possible de classer nos inputs en

Rapport-Gratuit.com

plus de deux outputs différents (non binaire) grâce à l'ajout de plusieurs couches cachées avec un nombre de neurones variable(L, 2018).

Ainsi, le fonctionnement du Perceptron qui reçoit des inputs et qui calcule un output est conservé, à la différence qu'ici les neurones d'une couche sont reliés à tous les neurones de la couche suivante. La couche cachée qui contenait uniquement un neurone précédemment, en contient désormais « n » et avec également un output possible. Voici une figure qui exprime les modifications apportées :

Figure 4 : Le Perceptron multi couches



(Wikipedia, 2018b)

Le traitement de chacun des neurones des couches cachées utilise toujours les inputs pondérés par des poids en entrée et y applique une fonction d'activation pour tous les neurones. Dans ce système, lorsque la fonction d'activation a donné un résultat qui est au-dessus du seuil d'activation définit, la connexion entre le neurone courant et les suivants se renforce. Dans le cas contraire, l'activation de la connexion entre le neurone courant et les suivants ne se fait pas et c'est ainsi, par propagation depuis les inputs, vers les outputs, que le Perceptron multi couches fonctionne. Cependant, il demeure le problème de propagation de l'erreur lorsque les résultats se rependent ainsi de couche en couche. C'est-à-dire qu'à chaque passage d'une couche « n » à une couche « n+1 », le réseau commet de légères erreurs qui, au fil du passage dans le réseau, s'agrandissent et risquent donc de fausser les résultats finaux.

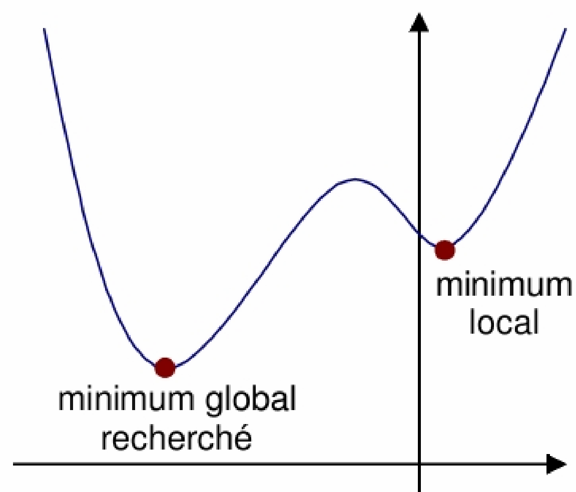
Ce phénomène peut être réduit par l'utilisation de l'algorithme du gradient qui nous permet de calculer l'erreur de la dernière couche vers la première. La rétropropagation de cet algorithme permet de converger itérativement vers un paramétrage des poids

synaptiques le plus optimisé possible.(Wikipedia, 2018e) Le fait d'utiliser cet algorithme de manière itérative constitue « l'entraînement » du réseau de neurones et nous en parlons dans la section « 5.2 Phase d'apprentissage ».

L'utilisation du gradient se fait par comparaison entre le résultat obtenu après l'utilisation de notre réseau de neurones avec le résultat attendu. L'algorithme est utilisé de manière itérative pour que cette différence diminue jusqu'à atteindre un minimum local, voire global (voir Figure 5). Le minimum global défini le minimum qu'il est possible d'atteindre globalement à toutes les applications de l'algorithme du gradient. On parle alors de gradient global.(Dupré, 2017) Alors que le minimum local définit le minimum qu'il a été possible d'atteindre pour l'instant mais qui pourrait ne pas être le plus bas niveau.

Voici une représentation graphique de la descente du gradient lorsqu'il est appliqué itérativement sur le résultat d'une fonction :

Figure 5 : Descente du Gradient



(Couchot, Guyeux, 2008)

On peut voir sur la Figure 5 : Descente du Gradient la courbe représentant une fonction d'erreur avec la descente du gradient représenté ici par le point brun qui descend le long de la fonction d'erreur pour y trouver le minimum. Une fois celui-ci trouvé, le réseau adapte les différents poids de chaque connexion pour arriver enfin au résultat qui minimise au maximum l'erreur.

Ainsi, le réseau devient de plus en plus précis dans ses prédictions du fait que l'erreur tend à diminuer et qu'il y a un renforcement des connections neurologiques qui offre des prédictions correctes et une diminution de l'importance des connexions dont résultent des prédictions erronées.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

2.3 Les variantes actuelles

2.3.1 CNN

2.3.1.1 Origine et utilisation actuelle

CNN qui signifie Convolutional Neural Network est un réseau de neurones à convolution. Il consiste à un empilage multicouche de perceptron dans le but de traiter des images et pour éviter d'avoir un nombre trop important de connexion et de poids synaptiques. (Doukkali, 2017) Par exemple, si une image a une taille de 1'024 pixels, ce qui fait 32 * 32 pixels et que la première couche cachée contient 128 neurones, ce réseau de neurones multicouche aurait 1024 entrées et 131'072 connexions synaptiques pondérées par des poids (1'024 * 128). Alors qu'un réseau de neurones à convolution va séparer une image en entrée en plusieurs images à l'aide d'une étape de convolution. Par conséquent, nous parlons d'empilage multicouche, car il y aura plusieurs couches qui auront différentes responsabilités comme la convolution ou le pooling.

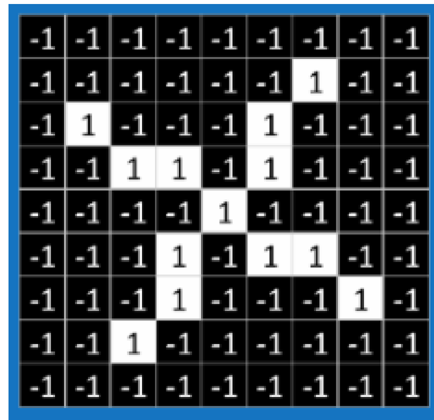
Il n'est pas nécessaire de détailler d'avantage chaque étape d'un réseau de neurones de type CNN dans le cadre de ce chapitre, puisqu'il s'agit ici de faire un état de l'art des différentes architectures existantes et d'en survoler leur fonctionnement. Nous entrerons dans les détails de l'architecture choisie après la rubrique « 2.4 Lequel choisir ? ».

Ce modèle est inspiré du fonctionnement du cortex visuel des animaux puisque celui-ci occupe le lobe occipital du cerveau et sert au traitement des informations visuelles et que les neurones de cette région se chevauchent lors du pavage du champ visuel.(Wikipedia, 2018d)

Dans le cadre de l'analyse d'une image par un réseau de neurones convolutif, il s'agit de découper celle-ci en petites zones, aussi appelés « tuiles » d'une taille que nous devons définir. Chacune de ses tuiles correspond à un input dans le réseau de neurones et sera donc traitée individuellement par un neurone.(Wikipedia, 2018d)

L'idée directrice du CNN est de ne pas comparer l'entièreté d'une image avec une autre pour savoir si elles sont similaires, mais de comparer certaines caractéristiques de celles-ci pour déterminer si elles contiennent des similitudes. Ceci permet d'avoir par exemple une image contenant un « X » :

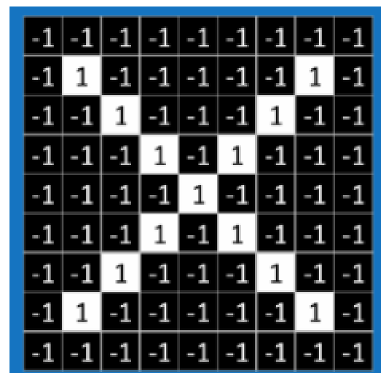
Figure 6 : Image d'un « X » décalé CNN



(Crouspeyre, 2017)

Et une image contenant un « X », mais de manière décalée :

Figure 7 : Image d'un « X » CNN

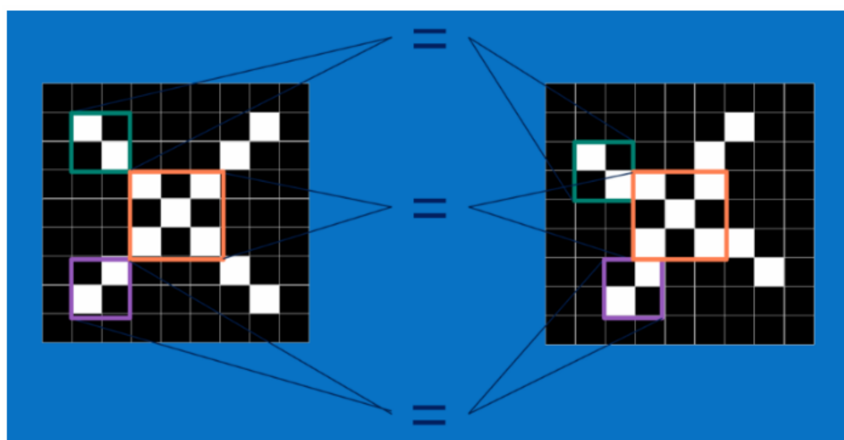


(Crouspeyre, 2017)

Et de pouvoir dire que dans les deux cas il s'agit de la lettre « X ». Un CNN va donc découper chacune des images en plus petits blocs, d'une taille que nous aurons définie, et tenter de repérer des blocs qui se ressemblent approximativement.

Dans le cas présenté entre la Figure 6 : Image d'un « X » CNN et la Figure 7 : Image d'un « X » décalé CNN, il va dégager des caractéristiques principales comme deux diagonales de la lettre « X » avec son entrecroisement qui sont les traits les plus communs de cette lettre.(Crouspeyre, 2017) Voici visuellement comment les caractéristiques vont être repérées et comparées par le réseau :

Figure 8 : Caractéristiques similaires CNN



(Crouspeyre, 2017)

Comme nous ne savons pas d'avance si certaines caractéristiques sont présentes ou non dans l'image donnée, le réseau va donc chercher dans l'entièreté de l'image, appelé aussi, filtrage. Cette phase consiste à, premièrement, définir un filtre pour chaque caractéristique que nous aimerions retrouver dans notre image, ainsi que sa taille. Un filtre se présentera, sous forme d'une matrice, de la manière suivante :

Figure 9 : Filtre CNN

6	8
3	4

(Duc-Jacquet, 2017)

Ce filtre est ensuite utilisé sur les 4 premiers pixels de la Figure 7, puis il est décalé et appliqué sur toute l'image. A chaque fois que le filtre est décalé sur l'image, on multiplie la valeur de chaque cellule du filtre avec la valeur du pixel sur lequel elle se trouve. Le résultat pour chaque application d'un filtre sera un vecteur qui contiendra à chaque indice le résultat de la multiplication de la valeur du pixel avec chacune des valeurs du filtre. Comme les caractéristiques que nous voulons repérer dans la Figure 8 ont des pixels qui ont une valeur plus élevée que les autres, le résultat de l'application d'un filtre sur ces parties-là sera plus élevé que sur des parties de l'image qui ne nous intéressent pas. Ainsi, le CNN n'aura plus qu'à chercher des résultats d'application de filtre sur les deux images qui sont très proches pour pouvoir affirmer qu'il s'agit, dans notre exemple, de deux images contenant un « X ».

Ce type de structure permet principalement l'application de la reconnaissance d'image et de vidéos, mais aussi du traitement du langage naturel de manière automatisée.

En ce qui concerne le traitement du langage, l'application du réseau de neurones convolutif est utilisé de la même manière que pour une image, mais ce ne sont pas des blocs d'une image qui sont passés en entrée, mais des parties de phrases par exemple.

Cependant, il existe un problème majeur à prendre en compte lorsqu'il est nécessaire de faire de l'analyse de texte avec un CNN. En effet, ce réseau repose sur le principe que les éléments d'un bloc, pixels pour une image et mots pour une phrase, sont sémantiquement liés. Or, ceci est très souvent vrai pour une image, alors que pour une phrase ce n'est pas toujours, voir pas souvent le cas. Prenons l'exemple d'un visage qui doit être repéré par notre réseau. Si celui-ci arrive à détecter un nez, des yeux et une bouche, il arrivera très probablement à nous dire qu'il s'agit d'un visage et sans avoir besoin de connaître les autres éléments de l'image en question. Si nous prenons l'exemple d'une phrase dans laquelle il faut déterminer le contexte, la corrélation entre un nom et son adjectif, CNN ne pourra que détecter quel type de mot est lié à un autre et déterminer qu'un adjectif se rapporte à un nom par exemple. Par contre, il sera très difficile pour lui de déterminer le sens que peut donner un adjectif une fois appliqué sur un nom.(Britz, 2015)

2.3.2 RNN

2.3.2.1 Origine et utilisation actuelle

RNN qui signifie Recurrent Neural Network est un réseau de neurones qui, contrairement aux réseaux de neurones convolutifs, transmet de l'information de l'étape actuelle à la suivante. Ainsi, il est plus proche du fonctionnement réel d'un réseau de neurones humain qui permet lui aussi de traiter une information courante selon les précédentes.

Le terme récurrent désigne le fait que le réseau ne possède qu'une seule couche cachée et qu'à chaque fois que cette couche reçoit une entrée et produit un résultat, celui-ci est retransmis à cette même couche, en plus de l'entrée suivante. Cette unique couche va permettre de traiter chaque nouvelle entrée en prenant en compte le résultat fournit à l'instant t-1 (*t-1 fait référence à l'étape précédent un instant donné, noté « t »*), qui lui-même a pris en compte le résultat à l'instant t-2, etc. Etant donné que la sortie de la couche à un instant t est calculée en fonction de tous les inputs passés

précédemment, on peut lui attribuer la caractéristique de « mémoire ». Cette caractéristique dans un réseau de neurones récurrent est appelée « cellule mémoire ».(Géron, 2017)

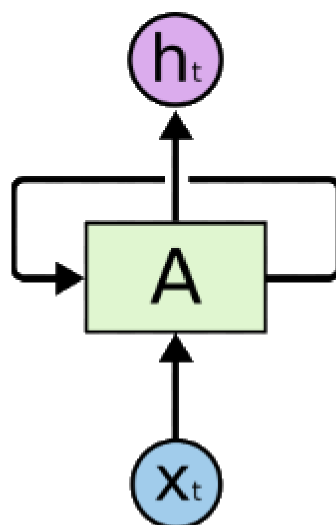
Ainsi, il est particulièrement adapté aux situations dans lesquelles est impliqué un certain contexte et lorsque les données forment une suite logique et sont dépendantes les unes des autres.(DataAnalyticsPost, 2018)

Il existe cependant un défaut à ce type de réseau de neurones qui est sa capacité limitée à enregistrer des éléments passés dans sa cellule mémoire pour traiter des éléments futurs. En effet, il s'agit d'une mémoire à court terme du fait que les éléments reçus à l'entrée de la couche soient modifiés par les éléments de la couche actuelle. Ainsi, à partir d'un certain nombre d'itération dans la couche cachée, il sera impossible de prendre en compte les premiers éléments précédemment enregistrés qui auront été écrasés par ses successeurs.

Nous pouvons prendre l'exemple d'un tel réseau utilisé pour générer la suite d'une collection de livres, tout en considérant les évènements passés dans les précédents tomes. Ce réseau récurrent sera capable, au fil des itérations, de prendre en compte chacun des éléments qui se sont déroulés dans l'intrigue des premiers livres, jusqu'à arriver au point où il commencera à perdre les premiers éléments et à les remplacer par les nouveaux. Ainsi, le nouveau tome généré pourrait probablement réciter des évènements qui se sont déjà produits au début de la collection.

La figure suivante démontre schématiquement le fonctionnement d'un tel réseau :

Figure 10 : Schéma RNN

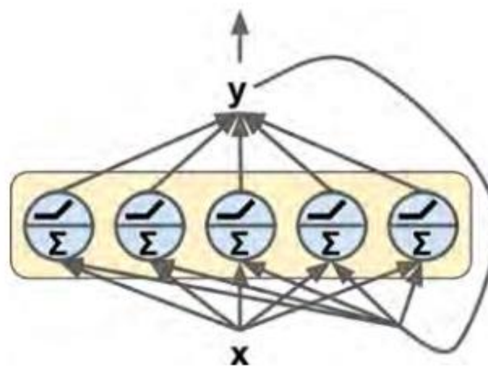


(Olah, 2015)

Ce schéma contient une couche d'inputs à un instant « t » et est représenté par la variable X_t, une couche cachée représentée par la variable A et une couche d'outputs représentés par la variable h_t. A la différence d'un réseau de neurones multicouches, on peut voir ici que la couche cachée a un lien vers elle-même et transmet des outputs en sortie de sa couche vers elle-même, en plus des inputs suivants qu'elle recevra en entrée. Cette architecture forme donc une boucle, ce qui différencie ce réseau puisqu'il permet d'avoir une mémoire des éléments précédemment traités.

Désormais, nous savons à quoi ressemble une architecture RNN et nous avons une intuition quant à son fonctionnement technique. Voici un schéma qui démontre de manière technique comment un tel réseau fonctionne :

Figure 11 Fonctionnement technique d'un RNN



(Géron, 2017)

Sur le schéma de la « Figure 11 », le « x » représente les inputs de notre réseau récurrent, le « y » représente les outputs et le rectangle représente la couche cachée avec sa multitude de neurones. Chaque input est relié à chacun des neurones de la couche cachée et chaque neurone de la couche cachée est relié aux outputs. Ces liens représentent les poids des connexions. De plus, la sortie de chacun des neurones de la couche cachée est obtenue ainsi :

Équation 1 : Sortie d'un neurone dans un RNN

$$y_{(t)} = \phi({x_{(t)}}^T \cdot w_x + {y_{(t-1)}}^T \cdot w_y + b)$$

(Géron, 2017)

Le neurone multiplie donc l'entrée, notée $x^{(t)}$, par le poids de la connexion correspondante, notée w_x, et y additionne la sortie obtenue à l'étape t-1, notée $y_{(t-1)}$, multipliée par le poids de la connexion correspondante, notée w_y, et y additionne le biais correspondant au neurone dans lequel on se trouve. Enfin, une fonction

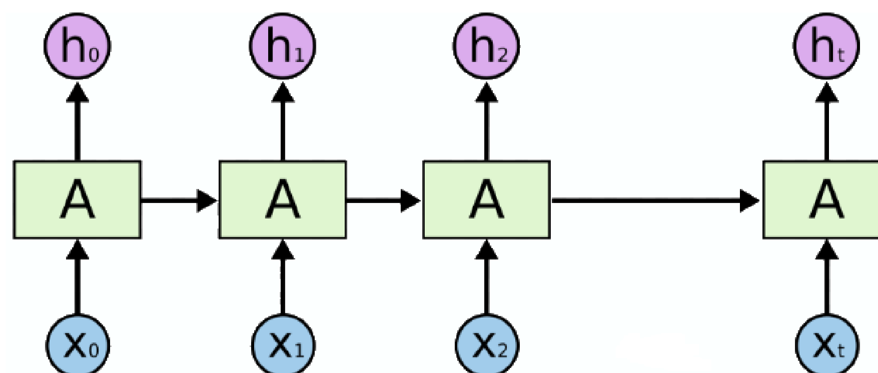
d'activation, notée ϕ, est appliquée sur ce résultat. Voici donc ce qu'effectuera chacun des neurones de la couche cachée, à chaque fois qu'une entrée et le résultat de l'étape t-1 lui sont fournis.

Comme chacun des neurones fournit un seul résultat en sortie, l'output global de tous les neurones de la couche cachée est représenté par un vecteur qui contient chacun des résultats de chaque neurone. Ainsi, si nous avons 12 neurones dans notre couche cachée, nous aurons, après chaque passage dans la couche cachée, un vecteur de sortie de taille 12, qui lui-même sera passé à l'étape suivante.

Aussi, on peut en ressortir que chaque neurone a deux types de poids : les poids des entrées à l'instant t et les poids des sorties à l'instant t-1. Les poids des sorties correspondent à ceux de l'instant t-1 puisqu'à chaque entrée passée, il s'agit de fournir le résultat obtenu à l'étape précédente.

Il est important de retenir que le schéma de la Figure 10, est une représentation à un instant t du fonctionnement d'un réseau récurrent. Alors que si nous voulons observer, de manière globale, l'état du réseau sur plusieurs instants, celui-ci se présenterait, de façon « dépliée », ainsi :

Figure 12 : Schéma RNN déplié



(Olah, 2015)

Ce schéma présente la même couche cachée dans différents états. Elle reçoit d'abord un input « X0 », puis calcule un output « H0 » et va le transmettre à elle-même. Ensuite, en fonction de « H0 » et du nouvel input « X1 », cette couche va fournir un nouvel output « H1 », etc.

Enfin, pour en revenir à la « cellule mémoire » qui est passée d'étape en étape dans notre réseau de neurones récurrent, son utilisation a génère un autre problème qui a longtemps mis un frein au développement des réseaux RNN qui est l'effet de disparition du gradient. Pour rappel, le gradient, résultat de l'application de l'algorithme

du gradient, représente le niveau auquel un poids synaptique doit être modifié pour minimiser l’erreur dans un réseau de neurones.

2.3.2.2 Vanishing / Exploding gradient

Nous avons abordé à la rubrique « 2.3.2.1 Origine et utilisation actuelle » le phénomène de disparition de gradient et nous allons ici en déterminer les différentes causes, ainsi qu’une solution proposée par des chercheurs :

La première cause de ce problème réside dans le fait qu’un réseau de neurones récurrent est formé de telle manière qu’il fonctionne comme une boucle qui reçoit à chaque itération le résultat de la couche précédente. Sans mesure particulière, il nous est impossible de préserver l’état de toutes les itérations précédentes lors de l’exécution de l’itération courante dans une boucle. Une première solution est de conserver en mémoire chacun des états de sortie, chacun des états d’entrée et une copie du réseau pour chacune des couches pour pouvoir ensuite remonter cette liste de la fin vers le début en calculant le gradient de l’erreur. Cette technique est appelée « Back Propagation Through Time ».(Brownlee, 2017a) Malheureusement, étant limité par la capacité en mémoire de la machine, cette solution devient très vite ingérable puisqu’elle est de plus en plus gourmande en mémoire en fonction du nombre d’étapes effectuées à travers la couche cachée.

La deuxième cause de la disparition du gradient de l’erreur, lorsque la méthode « Back Propagation Through Time » n’est pas utilisée, est le fait que lorsqu’il faut remonter dans les étapes du réseau récurrent, le gradient obtenu à chaque étape doit être passé à l’étape précédente pour tenter de remonter l’erreur jusqu’à sa source et de pouvoir modifier les poids synaptiques des neurones concernés. A chaque fois que le gradient est remonté d’une étape, il est en fait multiplié avec le suivant et ainsi de suite. Le problème étant qu’un gradient peut être soit supérieur à 1, soit inférieur à 1, alors si celui-ci est supérieur à 1, la valeur de correction globale ne cesse d’augmenter jusqu’à, à terme, causer un signal d’erreur infini. Dans le cas contraire où le gradient est inférieur à 1, celui-ci ne cesse de décroître jusqu’à atteindre 0 où la correction finie par disparaître probablement avant même que nous ayons réellement remonter l’erreur jusqu’à son origine. Dans le premier cas énoncé, il s’agit du phénomène « Exploding gradient » (explosion du gradient) et dans le deuxième cas, il s’agit du phénomène « Vanishing gradient » (disparition du gradient).(White, 2017)

Rapport-Gratuit.com

Pour pallier la limitation en mémoire de la machine et à la disparition du gradient, il existe une adaptation de la « Back Propagation Through Time » qui s'appelle « Truncated Back Propagation Through Time ». Ici, il est question d'effectuer une rétropropagation du gradient de l'erreur par séquence dont la taille est prédéfinie. C'est-à-dire qu'au lieu d'appliquer la rétropropagation à la fin d'un cycle d'entraînement de notre réseau récurrent, il faut le faire périodiquement et dans un intervalle que nous aurons défini en fonction du meilleur résultat obtenu. De ce fait, si la taille de la séquence est définie à cinq, la rétropropagation se fera toutes les cinq étapes et ne devra remonter que sur celles-ci. Cette méthode permet donc de régler le problème de limitation en mémoire de la machine. (Brownlee, 2017a)

2.4 Lequel choisir ?

Compte tenu des différentes qualités et des défauts de ces deux méthodes de construction de réseaux de neurones, il est nécessaire de lister ceux-ci pour chacun des deux types.

Pour les réseaux de neurones convolutifs, les défauts majeurs sont qu'ils ne parviennent pas à bien déterminer le contexte d'une phrase et qu'il ne possède pas de mémoire entre chaque passage à la couche suivante. Ceci fait que les CNN sont très performants lorsqu'il s'agit de tâches simples d'analyse de texte, mais que lorsqu'il s'agit de déterminer un mot manquant dans un phrase selon un contexte, ils le sont beaucoup moins.

Il n'y a donc pas la possibilité de conserver le résultat d'une couche pour le passer à la suivante qui pourra moduler le résultat qu'elle enverra à la suivante selon celui-ci.(Minz, 2016) Alors que le défaut majeur des réseaux de neurones récurrents est le temps de calcul qui est environ huit fois plus important comme le montre ce tableau :

Figure 13 : Comparaison CNN et RNN

Convolution Results

Input Size	Filter Size	# of Filters	Padding (h, w)	Stride (h, w)	Application	Total Time (ms)	Fwd TeraFLOPS	Processor
W = 700, H = 161, C = 1, N = 32	R = 5, S = 20	32	0, 0	2, 2	Speech Recognition	2.98	6.63	TitanX Pascal
W = 54, H = 54, C = 64, N = 8	R = 3, S = 3	64	1, 1	1, 1	Face Recognition	0.63	10.55	TitanX Pascal
W = 224, H = 224, C = 3, N = 16	R = 3, S = 3	64	1, 1	1, 1	Computer Vision	3.99	3.6	TitanX Pascal
W = 7, H = 7, C = 512, N = 16	R = 3, S = 3	512	1, 1	1, 1	Computer Vision	2.93	5.88	TitanX Pascal
W = 28, H = 28, C = 192, N = 16	R = 5, S = 5	32	2, 2	1, 1	Computer Vision	1.57	6.59	TitanX Pascal

Recurrent Ops Results

Hidden Units	Batch Size	TimeSteps	Recurrent Type	Application	Total Time (ms)	Fwd TeraFLOPS	Processor
1760	64	50	Vanilla	Speech Recognition	8.48	1.14	TitanX Pascal
2560	32	50	Vanilla	Speech Recognition	24.69	1.69	TitanX Pascal
1024	128	25	LSTM	Machine Translation	16.90	5.41	TitanX Pascal

(Dernoncourt, 2017)

On peut voir dans la colonne « Total Time » le temps d'exécution de chacun des types de réseaux de neurones. Nonobstant la différence du temps de traitement qui est minime et se compte en secondes, celle-ci ne serait décisive que si son utilisation serait faite en temps réel. Le réel atout d'un RNN réside donc dans le fait qu'il est capable de prendre en compte le contexte d'un contenu lors de sa classification, notamment dû à sa capacité de mémorisation entre chaque couche.(Minz, 2016) Dans le cadre de ce travail de Bachelor, nous opterons donc pour une architecture de réseau de neurones récurrente (RNN).

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

3. Long Short-Term Memory (LSTM)

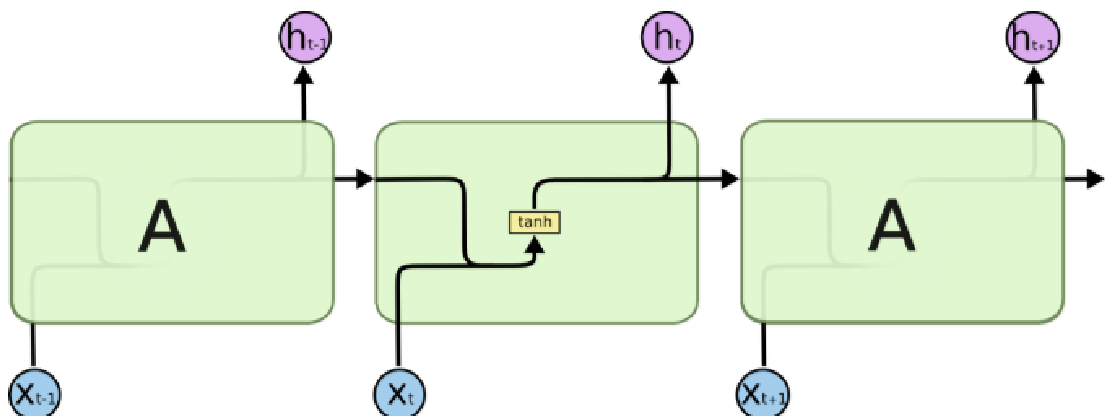
Ayant opté pour une architecture de type RNN et malgré les différentes solutions aux problèmes de celui-ci énoncées dans la rubrique «2.3.2 RNN », cette architecture ne permet toujours pas de pouvoir prendre en compte des informations passées sur les premières étapes aux dernières si notre réseau en comporte un nombre important. Il nous faut donc opter pour une spécialisation de réseaux de type RNN qui règle cette difficulté.

Ainsi, l'architecture des réseaux de neurones de type LSTM qui a été inventée en 1997 par Sepp Hochreiter et qui est une spécialisation de réseaux récurrents, règle la difficulté de mémoire à court terme des RNN. C'est-à-dire qu'ils sont construits pour résoudre des problèmes nécessitant un nombre important d'étapes. (Olah, 2015) Cela se fait avec l'introduction d'une « cellule d'état » qui se charge de garder en mémoire les éléments à travers les étapes, celle-ci peut être apparentée à la « cellule mémoire » d'un RNN. Ainsi, grâce à une structure qui gère l'ajout, la suppression et la modification de cette « cellule d'état », les réseaux LSTM ont donné une nouvelle perspective quant à l'utilisation de réseaux de neurones récurrents.

3.1 Fonctionnement

Pour en revenir au fonctionnement d'un RNN, abordé dans le chapitre « 2.3.2 RNN », chacune des couches cachées contient une seule fonction d'activation pour l'ensemble des neurones de celle-ci. Cette fonction va permettre d'affecter un seuil d'activation à chacun des neurones et celle-ci va affecter ensuite le poids synaptique des connexions à la couche suivante. La figure suivante montre un réseau de neurones récurrent avec une seule fonction d'activation « Tanh » par couche :

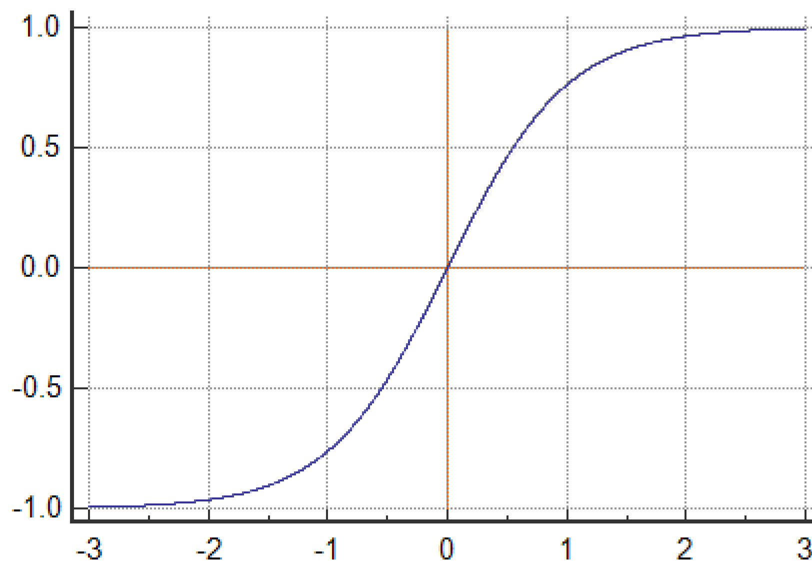
Figure 14 : Simple RNN



(Olah, 2015)

La tangente hyperbolique « Tanh » permet d'affecter un seuil d'activation à chaque neurone d'une couche et de modifier les poids qui vont découler de la sortie de ce neurone. « Tanh » se présente ainsi graphiquement :

Figure 15 : Tangent hyperbolique Tanh

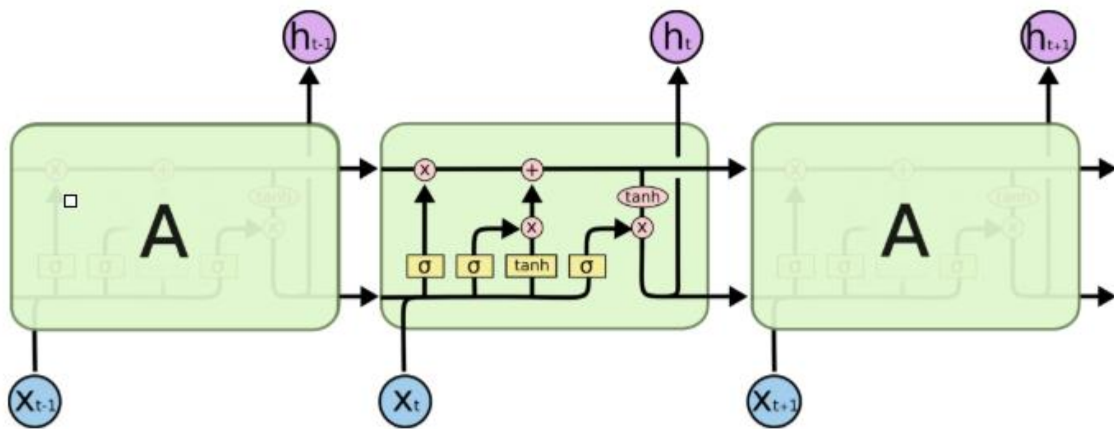


(Schoonjans, 2018)

Cette fonction va définir un seuil d'activation (axe Y) selon la valeur qui va être retournée par la somme des inputs pondérés par les poids (axe X) d'un neurone. Par exemple, si cette somme nous retourne un résultat de « 1 », la fonction « Tanh » retournera un seuil d'activation d'environ « 0.75 ». Ainsi, « Tanh » appliquera un seuil d'activation sur chaque neurone d'une couche et ceci permettra d'utiliser certains neurones de chaque couche lorsque des inputs passent dans le réseau de neurones.

L'architecture des LSTM, bien qu'étant un type de RNN, se présente plutôt de cette manière-là :

Figure 16 : Architecture LSTM



(Olah, 2015)

Entre la

Pour en revenir au fonctionnement d'un RNN, abordé dans le chapitre « 2.3.2 RNN », chacune des couches cachées contient une seule fonction d'activation pour l'ensemble des neurones de celle-ci. Cette fonction va permettre d'affecter un seuil d'activation à chacun des neurones et celle-ci va affecter ensuite le poids synaptique des connexions à la couche suivante. La figure suivante montre un réseau de neurones récurrent avec une seule fonction d'activation « Tanh » par couche :

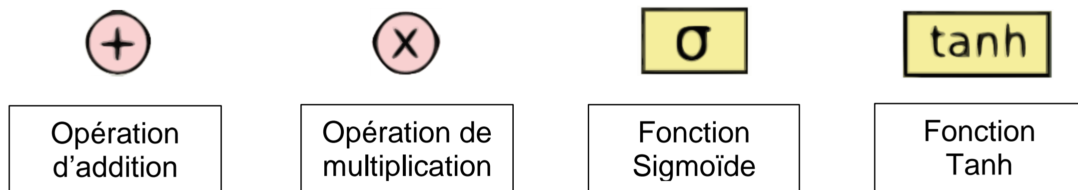
Figure 14 et la

Figure 16, on peut observer des points communs notamment au niveau des input (x_{t-1}, x_t et x_{t+1}) et des outputs (h_{t-1}, h_t et h_{t+1}). Cependant, la réelle différence réside dans le fonctionnement interne qui résulte par l'utilisation d'une simple fonction « Tanh » pour un RNN et d'une structure bien plus complexe pour un LSTM.

Voici une figure représentant la liste des symboles utilisés dans la

Figure 16 avec leur signification :

Figure 17 : Symboles schéma LSTM



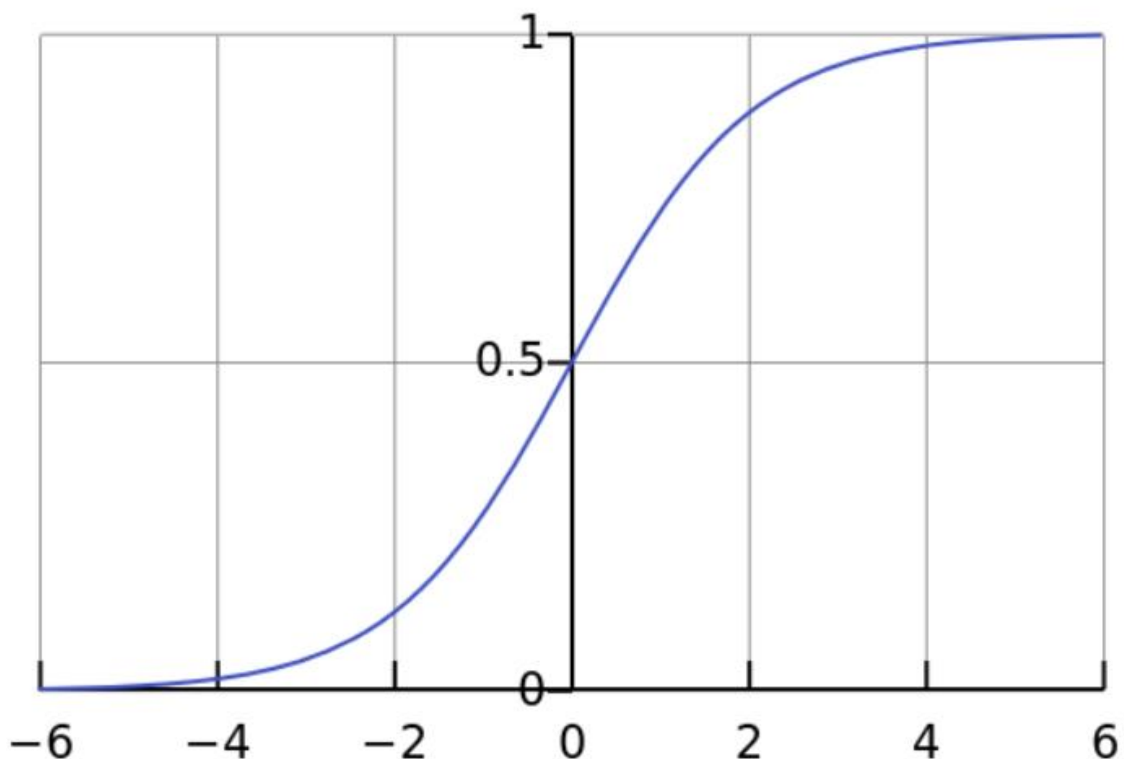
La caractéristique clé d'un LSTM est la ligne horizontale sur le haut de chaque couche du schéma de la

Figure 16. Celle-ci représente une cellule d'état qui est récupérée de la couche précédente et qui est susceptible d'être modifiée avant d'être passée à la couche suivante. Cette cellule est régulée par des structures qui sont appelées des « portes » et contient toutes les informations utiles de couche en couche. Il existe trois types de portes différents (Olah, 2015):

- La porte d'oubli
- La porte d'entrée
- La porte de sortie

La porte d'oubli est la première qui est appelée à l'entrée dans une couche d'un LSTM. C'est elle qui va définir quelles sont les informations de la cellule d'état récupérée qu'il faut garder et celles qu'il faut oublier. Pour ce faire, la porte va utiliser une fonction d'activation qui s'appelle Sigmoïde et qui fonctionne de la même manière que la tangente hyperbolique « Tanh ». Graphiquement, la fonction Sigmoïde se présente ainsi :

Figure 18 : Fonction Sigmoïde

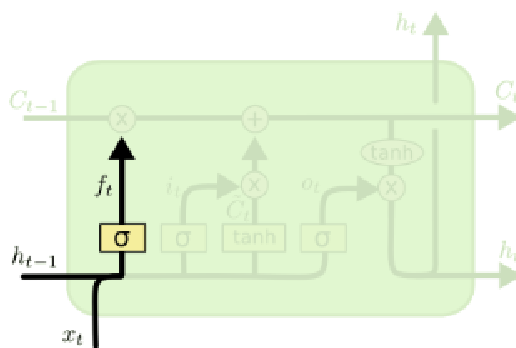


(Sharma, 2017)

La différence avec la fonction d'activation « Tanh » réside dans le résultat retourné qui sera compris dans l'intervalle [0 ; 1] pour la fonction Sigmoïde. L'avantage de l'utilisation de cette fonction est qu'elle permet de retourner « 0 » dans le cas où une information ne doit pas être gardée et un nombre compris entre « 0 » et «1 » dépendant de son importance. Alors qu'avec une fonction d'activation « Tanh », l'intervalle est plus grand et est compris entre « -1 » et « 1 ». « Tanh » pourrait donc être utilisée à la place de la fonction Sigmoïde, cependant ce qui nous intéresse dans la porte d'oubli est de pouvoir dire quelles informations il faut garder ou non.

Sur la figure suivante, nous pouvons observer que la porte d'oubli est appelée à l'entrée de la couche cachée, qu'elle reçoit en input X_t et H_{t-1} et que F_t va être l'application de la fonction Sigmoïde qui va influencer la cellule d'état reçue, ici Ct-1 :

Figure 19 : Porte d'oubli LSTM



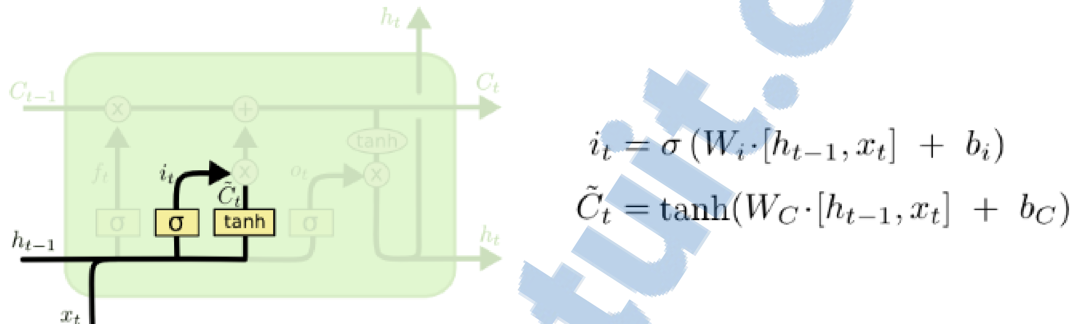
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

(Olah, 2015)

Le paramètre X_t représente les inputs reçus à l'entrée de la couche cachée, le paramètre H_{t-1} représente les outputs de la couche précédente reçus en entrée de la couche actuelle, le paramètre W_f est un vecteur contenant les poids des connexions entre la couche précédente et la couche actuelle et le paramètre B_f est le biais appliqué sur ce vecteur de poids. Le paramètre C_{t-1} représente la cellule d'état qui a été conservée de la couche précédente et qui est passé en entrée de la couche courante et enfin le paramètre F_t représente l'application de la fonction Sigmoïde sur les inputs et outputs reçus en paramètre de la couche actuelle. Le résultat de cette fonction va être compris dans l'intervalle [0 ;1] et celui-ci va être multiplié avec l'état passé en paramètre (C_{t-1}), ce qui va déterminer si celui-ci est gardé ou non.(Madhu Sanjeevi, 2018) Dans le cas où le résultat de la multiplication est nulle, cela indique que l'information n'est pas pertinente et qu'il ne faudra plus la prendre en compte dans les couches suivantes.

Une fois le passage par la porte d'oubli, il faut décider quelle information ajouter à l'état passé en paramètre. Pour cela, nous allons passer par deux étapes et la première de celles-ci est la porte d'entrée. Ces deux étapes sont présentées de manière schématique comme suit :

Figure 20 : Porte d'entrée LSTM



(Olah, 2015)

La porte d'entrée, comme la porte d'oubli, est représentée par une fonction Sigmoïde qui va être appliquée sur les inputs (X_t) et les outputs (H_{t-1}) fournis par la couche précédente. A la différence que dans l'application de la fonction Sigmoïde, ce sont les vecteurs de poids et le biais défini pour cette couche qui vont être appliqués et non pas les poids et le biais de la couche précédente utilisés dans la porte d'oubli. En parallèle à la porte d'entrée, nous allons créer un vecteur qui contient toutes les valeurs qu'il est possible d'ajouter à la cellule d'état par rapport aux inputs (X_t) et aux outputs (H_{t-1}) de la couche précédente, sans se soucier desquelles il faut garder. Ces valeurs sont le résultat de l'application de la fonction « Tanh » et sont donc comprises entre -1 et 1.

Une fois que les valeurs que nous voulons garder sont connues grâce à la porte d'entrée et que l'ensemble des valeurs qu'il est possible d'ajouter est établi sous forme de vecteur, nous allons multiplier le résultat de ces deux étapes ensemble. Le résultat de la multiplication est ensuite additionné avec la cellule d'état pour nous permettre d'ajouter de l'importance à certaines informations ou d'en enlever. L'étape de multiplication entre le résultat de la fonction «Tanh» et le résultat de la fonction Sigmoïde est obligatoire pour nous permettre d'avoir des informations avec une valeur de 0 dans la cellule d'état. Comme l'ajout d'information dans la cellule d'état se fait par une addition et que la fonction Sigmoïde nous retourne des résultats entre 0 et 1, il est impossible, même en ajoutant 0 à une valeur qui est plus grande que 0, d'avoir un résultat nul.(Kang, 2017) Ainsi, avec la multiplication du résultat de la fonction «Tanh» qui peut être négatif, il est possible de diminuer une valeur dans la cellule d'état jusqu'à atteindre une valeur nulle. Lorsqu'une information est à zéro dans la cellule d'état, elle

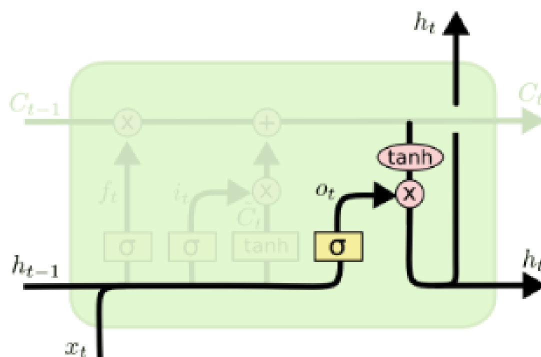
Rapport-Gratuit.com

doit être ignorée dans les étapes suivantes puisqu'elle est considérée comme non pertinente.

La dernière étape d'une couche d'un LSTM est la porte de sortie. Celle-ci est la partie suivante du schéma de la

Figure 16 :

Figure 21 : Porte de sortie LSTM



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

(Olah, 2015)

La porte de sortie va définir quelles sont les informations que la couche courante transmet à la couche suivante (H_t). Cette étape se divise en deux traitements principaux :

- Appliquer la fonction Sigmoïde
- Appliquer la fonction «Tanh»

Le premier traitement va appliquer la fonction Sigmoïde sur les inputs et les outputs transmis par la couche précédente, transformant les informations dans un intervalle [0 ; 1] selon l'importance de chacune d'entre elles.

Le deuxième traitement va appliquer la fonction «Tanh» sur la cellule d'état. Ensuite on multiplie le résultat des deux fonctions pour que les informations qui ont moins d'importance et qui sont contenues dans le résultat de la fonction Sigmoïde, puissent voir leur valeur diminuer. Ainsi, la porte de sortie ne transmettra à la couche suivante seulement les informations qui lui ont semblé être importantes en plus de la cellule d'état.

Un élément intéressant dans la structure d'un LSTM avec des portes est que si la porte de sortie d'une couche a considéré que certaines informations n'avaient pas d'intérêt à être passées à la couche suivante, celle-ci conserve tout de même ces informations dans la cellule d'état et peut potentiellement décider qu'une information est devenue importante et donc de la transmettre via sa porte de sortie à la couche suivante.

3.2 Particularités et paramètres

Cette section a pour but d'explorer les différentes particularités d'un Long-Short Term Memory (LSTM), notamment le format que les données qui lui sont passées vont

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

devoir respecter. Mais également les différents paramètres avec lesquels il nous est possible d'interagir pour pouvoir avoir un réseau qui soit le plus efficace et le plus personnalisé possible par rapport à notre cas d'utilisation.

3.2.1 Word embeddings

La principale particularité d'un réseau de neurones de type LSTM lorsqu'on l'utilise pour faire du traitement du langage naturel est qu'un vecteur contenant des données sous forme numérique avec chaque indice de ce vecteur qui correspond à un neurone de la couche d'inputs, doit être utilisée en entrée. Or, dans notre cas, le vecteur correspond à une phrase et chaque indice représentera soit un mot, soit un caractère de ponctuation ou d'échappement (p. ex : un espace, une virgule, un point…)

Le problème est que la couche cachée qui suit la couche d'inputs doit utiliser des fonctions de type Sigmoïde et «Tanh» qui ne peuvent s'effectuer que sur des nombres. Or, nous passons des vecteurs de mots ici. La solution est d'utiliser une technique qui s'appelle « Word embedding ». Celle-ci va nous permettre de transformer ces vecteurs de mots en vecteurs de nombre réels. Ainsi, notre réseau de neurones sera capable d'utiliser des fonctions mathématiques sur les données passées en entrée.

Pour ce faire, plusieurs procédés différents existent et sont utilisés selon la tâche qu'il nous est nécessaire d'accomplir :

- Encodage One-hot
- TF-IDF (Term Frequency-Inverse document Frequency)
- Word2Vec
- GloVe (Global Vectors for Word Representation)
- FastText
- Poincaré embeddings

(Olejnik, 2017)

Chacun des procédés cités ci-dessus sera abordé et détaillé dans les sections qui suivent.

3.2.1.1 Encodage One-hot

Il s'agit ici de transformer un attribut qui peut prendre autant de valeurs qu'il existe de catégories différentes. Si nous avons par exemple un jeu de données avec des voitures qui ont toutes un attribut « Marque », celui-ci pourra prendre autant de valeurs différentes qu'il existe de marques de voitures sur terre. Dans le cadre du Machine Learning, ceci pose deux problèmes puisque comme énoncé dans la section « 3.2.1 Word embeddings », un réseau de neurones doit avoir des nombres en entrées. Dans l'exemple cité, il suffirait de transformer les valeurs textuelles de l'attribut « Marque » en représentations numériques. Nous aurions par exemple les valeurs « 0 », « 1 » et « 2 » qui seraient des valeurs de l'attribut « Marque » et qui correspondraient aux marques suivantes :

- Honda
- Mazda
- Opel

Cependant, cette solution nous permettrait seulement d'avoir un réseau de neurones fonctionnel, mais qui ne pourrait être performant dans ses prédictions et c'est en ceci que réside le deuxième problème. En effet, comme vu dans la section « 3.1 Fonctionnement », un LSTM utilise des fonctions mathématiques qui retournent des valeurs en fonction de la somme des entrées (inputs) pondérée par les poids. Ainsi, si dans notre exemple nous avons la 100ième valeur de l'attribut « Marque » qui prend la valeur « 99 », alors une voiture ayant cette valeur aura beaucoup plus d'importance en comparaison à une voiture ayant la 1ère valeur de l'attribut « Marque » qui prendrait la valeur « 0 ». En effet, la valeur « 99 », même avec un poids de « 2 », aurait beaucoup plus d'importance que la valeur « 0 » avec un poids de « 200 », puisque 99 * 2 > 0 * 200.(Vasudev, 2017) Avec cette technique, notre réseau de neurones perdrait tout l'utilité des poids synaptiques puisque la valeur des attributs serait déterminante et non plus les poids des connexions entre neurones.

L'encodage One-hot propose de convertir chacune des valeurs uniques d'un attribut en un attribut avec des valeurs binaires. C'est-à-dire que chaque nouvel attribut généré ne pourra prendre que deux valeurs possibles :

- 0 (cet attribut ne correspond pas à cette instance)
- 1 (cet attribut correspond à cette instance)

Ainsi, les valeurs « Opel », « Mazda » et « Honda » de l'attribut « Marque » de notre exemple constitueraient des attributs à part entière avec soit la valeur « 0 » pour dire

que la voiture ne correspond pas à cette marque, soit la valeur « 1 » pour dire qu'elle correspond à cette marque. (Hapnes Strand, 2016)

3.2.1.2 TF-IDF (Term Frequency-Inverse document Frequency)

Cette technique consiste à donner une valeur à un mot par rapport aux nombres d'occurrences de celui-ci dans l'ensemble d'un jeu de données. Plus un mot a d'occurrences, plus son score sera important et plus il aura de pertinence lexicale. La formule appliquée dans ce procédé est la suivante :

Figure 22 : Formule TF-IDF

$$w_{x,y} = tf_{x,y} \times \log\left(\frac{N}{df_x}\right)$$

(Fily, 2015)

Le terme $tf_{x,y}$ défini la fréquence du mot dans l'ensemble des données, le terme df_x défini la quantité de documents qui contienne le mot et N représente le nombre de documents. Dans un contexte d'analyse de phrases qui ne tient pas compte du document dans laquelle elles apparaissent, df_x défini la quantité de phrases qui contienne le mot en question. Il s'agit de multiplier la fréquence d'apparition d'un mot, par rapport à tous les mots, avec le ratio de documents dans lesquels le mot apparait. Pour chaque mot, nous obtenons donc un nombre réel qui représente leur importance par rapport à l'ensemble de données.

Ce procédé s'utilise généralement sur des documents ou phrases qui ont un lien sémantique entre eux. Largement utilisé dans le fonctionnement des moteurs de recherche lors du référencement d'un site web, il permet d'accorder de l'importance à certains termes en comparaison à d'autres. La recherche faite dans un moteur de recherche retournera les résultats qui fournissent la plus haute importance lexicale pour cette recherche dans leur site web en premier.

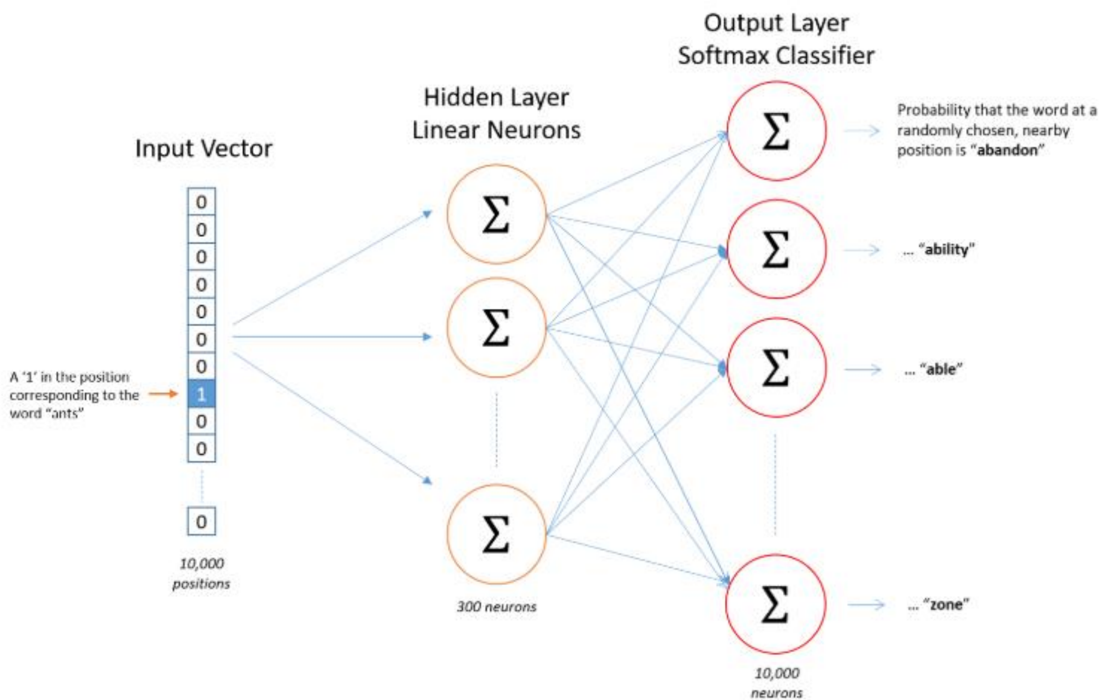
3.2.1.3 Word2Vec

Le principe émanant de Word2Vec, développé par Tomas Mikolov en 2013 et travaillant chez Google (Mikolov et al., 2013), est de transformer un mot en vecteur

numérique pour que celui-ci soit exploitable par un réseau de neurones artificiels. Ce vecteur ne contiendra pas seulement une représentation du mot, mais également les mots qui lui sont sémantiquement proches. Ceci permet de pouvoir détecter les relations entre les différents mots.

Word2Vec est en fait un réseau de neurones contenant une couche d'entrée, une couche cachée et une couche de sortie. La couche cachée ne contient que des fonctions de somme et c'est la couche de sortie qui contient une fonction d'activation.(Madhu Sanjeevi, 2018). Le modèle d'architecture utilisé pour ce réseau de neurones est le « Skip-gram ». Celui-ci a pour objectif initial de prédire les mots environnants d'un mot donné en entrée dans une phrase ou un document.(Mikolov et al., 2013) Voici un exemple graphique de l'architecture « Skip-gram » :

Figure 23 Exemple utilisation Skip-gram



(McCormick, 2016)

La couche d'entrée représentée par un vecteur sur la Figure 23 est constituée de « 0 » et de « 1 ». Ceci est dû à l'utilisation de l'encodage « One-hot », expliqué à la section « 3.2.1.1 Encodage One-hot », qui transforme ici un vecteur de mots en vecteur de « 0 » et de « 1 » pour chaque mot. Si nous voulons donc passer, comme dans l'exemple de la Figure 23, une liste de 10'000 mots dans ce réseau de neurones artificiels déjà entraîné et que nous voulons que celui-ci nous retourne les mots « proches » du mot « ants » parmi ces 10'000, nous devons lui donner un vecteur d'une taille de 10'000 avec des « 0 » et des « 1 », comme l'encodage One-hot le

préconise. Chaque indice du vecteur contiendra un « 0 » excepté l'indice du mot « ants » qui contiendra un « 1 » et qui représentera la position du mot en question. Après avoir appliqué la fonction de somme de l'un des neurones de la couche cachée, la sortie correspondra au(x) mot(s) qui la plus haute probabilité d'être proche de celui donné en entrée. Cette sortie prendra la forme d'un vecteur d'une taille de 10'000 avec à chaque indice la probabilité que le mot correspondant à cet indice soit proche du mot « ants » donné en entrée. Si nous voulons les cinq mots les plus proches, il nous suffit de prendre les cinq plus grandes probabilités sur les 10'000 fournies en sortie.(McCormick, 2016)

Le grand avantage à l'utilisation du réseau de neurones Word2Vec est que chacun des mots qui doit être converti en valeur numérique est également retourné avec des mots qui lui sont sémantiquement proches. Il sera donc plus facile dans un traitement à postériori de pouvoir comparer la sémantique de deux phrases, par exemple.

3.2.1.4 GloVe (Global Vectors for Word Representation)

Glove est un algorithme d'apprentissage non supervisé qui permet d'obtenir, comme Word2Vec, des vecteurs de représentations de mots. Cet algorithme a été développé en 2014 par Jeffrey Pennington, Richard Socher et Christopher D. Manning.(Pennington et al., 2014) Il s'agit d'un algorithme non supervisé dans le sens où nous n'avons pas besoin de lui fournir un corpus d'entraînement avec lequel il va adapter ses prises de décision pour s'améliorer avant qu'on puisse lui fournir le corpus cible.

Cet algorithme est basé sur l'agrégation statistique de co-occurrence de mots au sein d'un corpus. C'est-à-dire qu'il va repérer des mots qui apparaissent ensemble et à chaque fois qu'il va détecter une co-occurrence, il va incrémenter la valeur correspondante dans une matrice. Une co-occurrence pouvant concerner plus de deux mots, la matrice que nous devons construire aura autant de dimensions que la co-occurrence contenant le plus de mots. Voici un exemple de matrice de co-occurrence en deux dimensions et basé sur la phrase suivante « I love Programming. I love Math. I tolerate Biology. » (Vargas, 2017) :

Figure 24 Matrice de co-occurrence

	I	love	Programming	Math	tolerate	Biology	.
I	0	2	0	0	1	0	2
love	2	0	1	1	0	0	0
Programming	0	1	0	0	0	0	1
Math	0	1	0	0	0	0	1
tolerate	1	0	0	0	0	1	0
Biology	0	0	0	0	1	0	1
.	1	0	1	1	0	1	0

(Vargas, 2017)

Une fois la matrice effectuée, on peut placer les différentes co-occurrence dans un espace multidimensionnel tel qu'un graphique. Ainsi, certaines co-occurrence seront proches les unes des autres, voire au même endroit. On peut voir sur la Figure 24 que « Programming » et « Math » partagent les mêmes valeurs et seront placés au même endroit sur le graphique, ce qui signifie que dans ce contexte, ils sont très proches. Quant au terme « Biology », il sera le terme le plus proche de « Math » et de « Programming » au niveau des valeurs de co-occurrence qu'il a obtenu, ce qui signifie qu'il est très proche, dans ce contexte, au niveau du sens. Dans cet exemple simple de co-occurrence, nous analysons le 1er plus proche voisin dans le but d'avoir une matrice qui soit le plus simple possible à comprendre. Cependant, il est très commun d'en analyser les 5 plus proches, ce qui fait que cette matrice aurait 5 dimensions et serait visuellement très compliquée à analyser.(Vargas, 2017) Le problème avec cette technique est que la matrice grandi avec la taille du corpus et les calculs sont quadratiquement plus nombreux.

Après que la matrice de co-occurrences ait été créée, GloVe utilise une méthode de réduction des dimensionalités qui s'appelle « Singular Value Decomposition ». Cette méthode permet de réduire la taille de la matrice pour permettre d'accélérer l'accès à celle-ci en supprimant, par exemple, les mots qui apparaissent très rarement et en diluant la fréquence trop élevée de certains mots très communs comme les déterminants.(Ruder, 2016) Nous n'allons pas détailler cette technique dans le cadre de ce travail de Bachelor.

3.2.1.5 FastText

FastText est une librairie développée par le laboratoire de recherche dans l'intelligence artificiel de Facebook et rendue open-source en 2016. (Bojanowski et al., 2016) Cette librairie est basée sur le fonctionnement de Word2Vec, expliqué dans la section « 3.2.1.3 Word2Vec », mais la plus petite unité traitée avec FastText sera un « gram » et non pas un mot. Ainsi, il va traiter chaque mot comme étant composé de « n-grams » caractères. Un « gram » est une séquence contiguë de plusieurs caractères dont sa taille est généralement définie, dans le cadre de l'utilisation de la librairie FastText, entre 3 et 6 caractères. Si nous prenons l'exemple du mot « sunny », il sera composé de vecteur tel que :

- [sun, sunn, sunny]
- [sunny, unny, nny]

Cette représentation des mots fournie par FastText permet de gérer, avec une plus grande efficacité, des cas où un mot n'apparaît que peu ou pas du tout dans un corpus. Puisque chaque mot est découpé en « n-grams », il sera possible de trouver des mots qui puissent se rapprocher d'un « gram » et ceci augmente donc la possibilité de faire des liens. Alors qu'avec Word2Vec, un mot composé de plusieurs et qui n'a jamais été aperçu dans le corpus, ne parviendra pas à être lié à un ou plusieurs autres mots. Par exemple, si nous avons un mot « stupedofantabulouslyfantastic », FastText le découpera en plusieurs « gram » et cela permettra de le lier aux mots « fantastic » et « fantabulous », par exemple. (NSS, 2017)

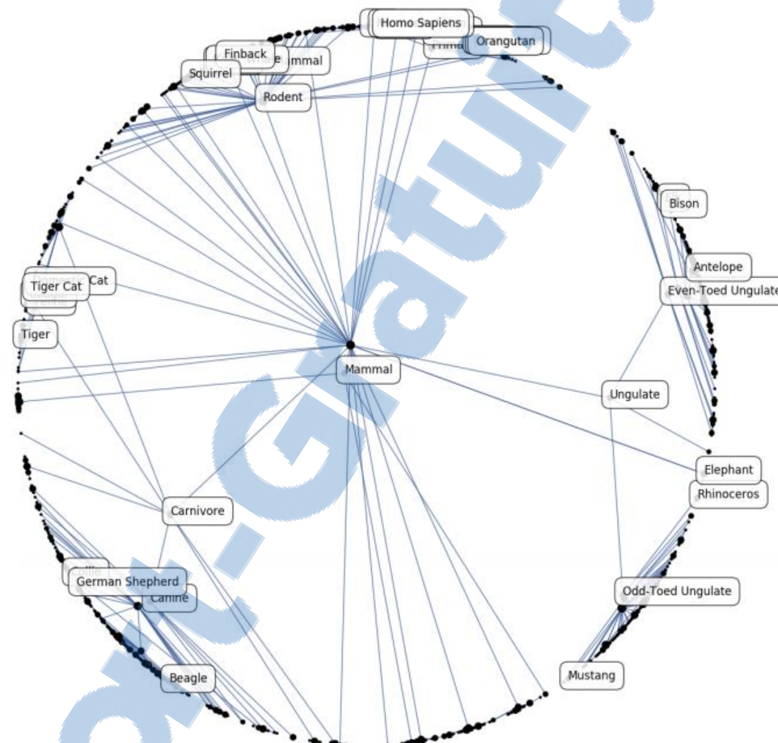
Une fois les mots découpés en « n-grams », le fonctionnement de FastText repose sur l'utilisation du réseau de neurones artificiels de Word2Vec, que ce soit avec l'architecture « Skip-gram » ou « C-bow ».

3.2.1.6 Poincaré embeddings

Ce modèle a été créé par le laboratoire de recherche dans l'intelligence artificiel de Facebook en 2017. (Nickel, Kiela, 2017) Le but de cette méthode consiste à diminuer la complexité de la représentation des relations entre les mots dans le cadre de l'analyse du langage naturel. En effet, plus le corpus à analyser est important et plus les liens entre les différents termes sont importants et plus la complexité de la représentation de ses liens augmente. Avant que Nickel Maximilian et Kiela Douwe ne

publie un document sur la méthode Poincaré, la plupart des modèles utilisaient la distance Euclidienne entre deux termes pour savoir si ceux-ci avant un lien sémantique. Leur méthode repose cependant sur un espace hyperbolique dans lequel les différents mots sont placés selon le lien sémantique qu'ils ont entre eux.(Nickel, Kiela, 2017) Voici un exemple de l'espace hyperbolique contenant quelques exemple de mots :

Figure 25 Exemple d'espace hyperbolique de la méthode Poincaré



(Nickel, Kiela, 2017)

Ceci permet donc une représentation des liens sémantiques tout en ayant un nombre de dimensions limité à cet espace hyperbolique. De plus, cet espace permet également de représenter une hiérarchie entre les mots, ce que ne permet pas un espace Euclidien puisque dans un tel espace, les mots sémantiquement liés seront simplement regroupés. Dans un espace hyperbolique, en plus d'être rassemblés, les mots se réfèrent toujours à un mot qui est hiérarchiquement plus haut et qui représente donc un concept plus général du mot courant jusqu'à remonter à la racine même.(Nickel, Kiela, 2017)

3.2.1.7 Choix

Après avoir listé et expliqué les différentes méthodes qui existent pour transformer un vecteur de mots en vecteur numérique avec pour chaque mot une représentation des

Rapport-Gratuit.com

liens qui peuvent exister avec d'autres termes, nous allons devoir en choisir une sur la base d'une comparaison des applications possibles et surtout des avantages de chacune d'elles.

3.2.1.7.1 TF-IDF vs Word2Vec

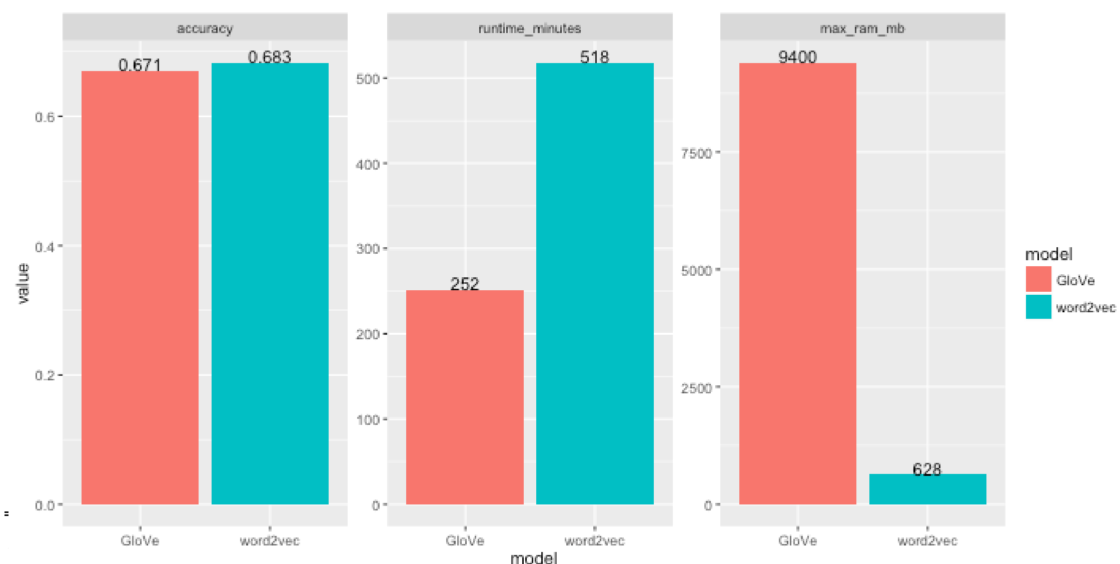
La méthode TF-IDF permet de donner une importance aux différents mots selon leur fréquence d'apparition et est surtout utile pour le référencement d'un site internet, par exemple. De plus, lors de son application, les résultats retournés sont des scores et ne permettent donc pas de faire des liens sémantiques entre différents mots, mais seulement de déterminer leur importance en termes de fréquence, au sein d'un corpus. Word2Vec permettant, à partir d'un mot donné en entrée, de générer un vecteur qui contient le mot et ceux qui lui sont sémantiquement liés, est très adapté à la tâche que nous aimerions accomplir. En effet, rappelons que dans le cadre de ce travail de Bachelor, l'intérêt est de pouvoir déterminer à quel point deux phrases sont sémantiquement liées. Entre ces deux méthodes, nous opterons donc pour Word2Vec qui va donc être comparée à la méthode suivante de la rubrique ci-dessous.

3.2.1.7.2 Word2Vec vs GloVe

Ces deux procédés étant très similaires et GloVe se basant sur le fonctionnement de Word2Vec, nous allons donc en choisir un par rapport aux différents résultats obtenus sur plusieurs tâches différentes.

Voici un premier graphique qui exprime certaines différences notamment au niveau de la vitesse d'entraînement, de la précision des prédictions et du coût en mémoire RAM des deux méthodes. Ces tests ont été exécutés sur la même machine et avec le même jeu de données :

Figure 26 Comparaison coût, RAM, temps de Word2Vec et GloVe

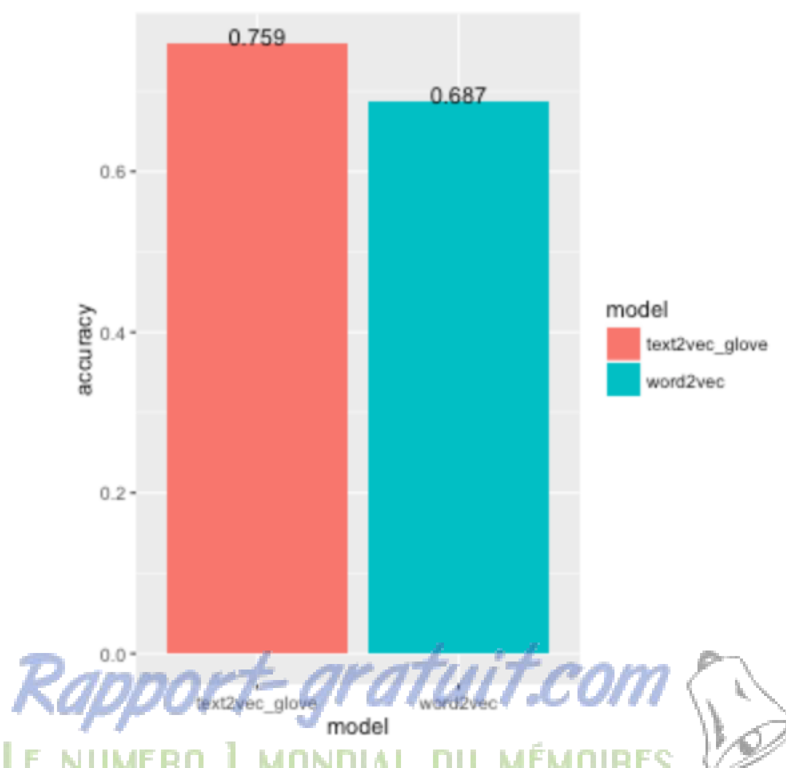


(Selivanov, 2015)

Nous peut voir que la précision des deux méthodes diffère de seulement ~1%, que le temps d'exécution de l'entraînement est à peu près deux fois moins long pour GloVe et que la RAM utilisé par Word2Vec est plus de 10 fois inférieure à la RAM utilisée par GloVe. La précision qui ne diffère que peu est due au fait que les deux procédés sont quasiment similaires au niveau du fonctionnement. La différence de temps d'exécution est dû au fait que GloVe utilise une matrice de co-occurrences pour rechercher des mots qui sont sémantiquement proche de celui donnée en entrée alors que Word2Vec comparera le mot en entrée avec tout le jeu de données. Enfin, la différence en consommation de mémoire vive est due au fait que GloVe stocke l'entièreté de sa matrice dans la mémoire RAM pour pouvoir y accéder plus rapidement alors que le jeu de données auquel doit accéder Word2Vec n'est pas stocké dans la mémoire vive, ceci explique également la différence au temps d'exécution. En effet, il est bien plus rapide d'accéder à la mémoire vive (RAM) sur un ordinateur qu'à la mémoire de stockage.

On peut voir sur ce deuxième graphique qu'avec quelques modifications sur les paramètres de fonctionnement des deux méthodes, GloVe est plus précis que Word2Vec, en plus d'être plus rapide :

Figure 27 Comparaison précision GloVe vs Word2Vec



Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

(Selivanov, 2015)

Après ces différents constats, nous allons opter pour la méthode que propose GloVe dans le cadre de ce travail de Bachelor. Puisqu'il est plus performant et plus rapide que la méthode proposée par Word2Vec. Concernant la grande quantité de mémoire vive consommée par GloVe, il existe des méthodes qui tendent à la réduire jusqu'à environ 50% de sa consommation de base.(Selivanov, 2015) Par exemple, la méthode créée par Google qui s'appelle MapReduce nous permet de modifier la structure de notre matrice en matrice avec des paires de « Key, value ».(Dean, Ghemawat, 2008) Nous ne détaillerons l'effet positif qu'a cette méthode sur la consommation de RAM dans ce chapitre puisque nous n'avons pas encore décidé si GloVe était la méthode la plus adaptée à ce problème.

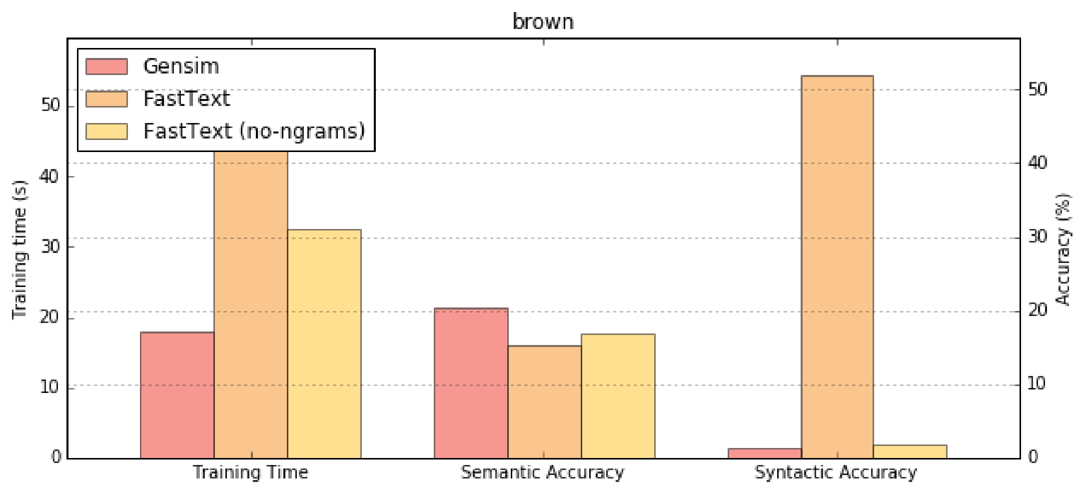
3.2.1.7.3 GloVe vs FastText

La différence fondamentale entre ces deux procédés réside dans le fait que FastText découpe chacun des mots en « n-grams » et qu'il serait plus à même de capturer les mots qui sont morphologiquement très proches. Par exemple, le mot « faster » serait probablement lié aux mots « fast » et « fastest » du fait que le gram « fast » se retrouvera dans chacun de ses trois mots. A la différence de GloVe, où l'on s'attend plutôt à avoir un mot qui sera lié de manière sémantique à un autre, FastText liera des mots plutôt par leur syntaxe.

Nous allons vérifier cela à l'aide de plusieurs graphiques mettant en comparaison le temps d'entraînement, la précision sémantique et syntaxique de FastText et de Word2Vec sur plusieurs corpus. Nous n'aurons pas l'occasion de comparer directement GloVe avec FastText, mais plutôt Word2Vec avec FastText puisque je n'ai pas pu pu trouver des résultats de comparaison entre GloVe et FastText. En outre, le but de ce travail de Bachelor n'étant pas de créer des jeux de tests entre les différents procédés de « Word embeddings », mais plutôt d'en sélectionner le meilleur, par rapport à notre cas d'utilisation. De plus, GloVe et Word2Vec sont très similaires au niveau de leur fonctionnement, hormis le fait que GloVe utilise une matrice de co-occurrences, qu'il est plus rapide à l'entraînement et plus précis dans ses prédictions.

Voici le premier graphique par rapport aux résultats obtenus sur le corpus « Brown », créé en 1961 et contient environ 1 million de mots anglais(Winthrop, Kucera, 1979) :

Figure 28 Word2Vec vs FastText sur le corpus « Brown »

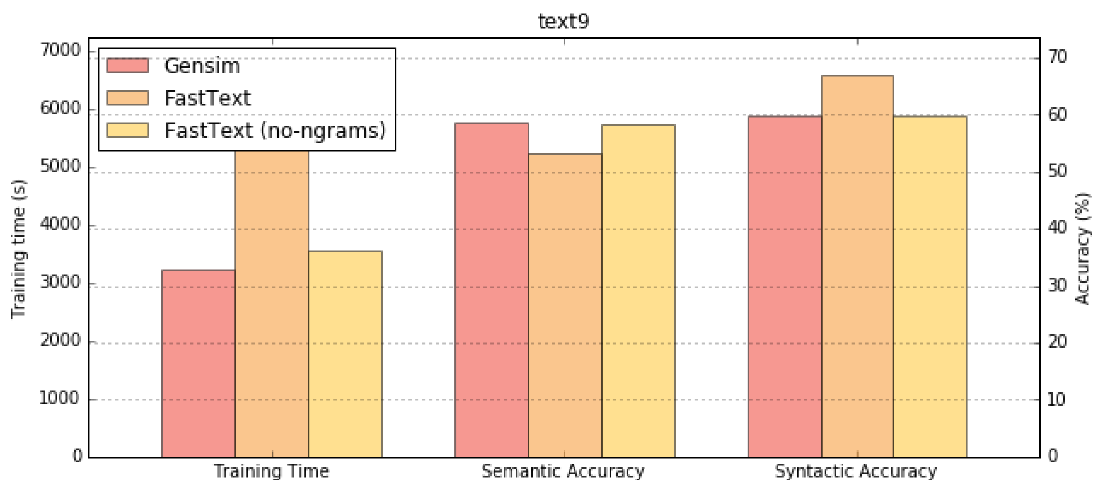


(Jayant, 2016)

On peut observer que sur un petit corpus, la librairie Gensim qui implémente Word2Vec est plus rapide et parvient à mieux déceler la sémantique des termes que FastText, lors de la phase d'entraînement. Cependant, FastText parvient largement mieux à regrouper les mots qui sont syntaxiquement proches.

Nous allons maintenant observer les résultats d'un test sur un corpus beaucoup plus conséquent pour voir l'évolution de ceux-ci. La taille du corpus « text9 » sur lequel a été effectué le test est d'environ 124 million de mots anglais (Roxor, 2016) :

Figure 29 Word2Vec vs FastText sur le corpus « text9 »



(Jayant, 2016)

On peut observer ici que FastText et Word2Vec fournissent des résultats beaucoup plus concluants. On peut donc imaginer que si le corpus d'entraînement était encore

plus conséquent, les résultats seraient progressivement plus intéressants. Les conclusions de ces résultats restent cependant les mêmes que sur un petit jeu de données comme « Brown », mais l'écart entre les deux méthodes est réduit, notamment au niveau de la capacité de Word2Vec à repérer les similitudes syntaxiques.

Par ailleurs, la capacité à déceler les similitudes sémantiques de Word2Vec reste supérieure à FastText et comme démontré dans la rubrique « 3.2.1.7.2 Word2Vec vs GloVe », GloVe obtient de meilleurs résultats concernant la précision que Word2Vec, on peut donc imaginer que celui-ci obtiendrait une précision à capter les similitudes sémantiques conséquemment plus grande que FastText.

Après ces différents constats, nous opterons donc pour GloVe et non pas FastText, puisque dans ce travail de Bachelor nous aurons essentiellement besoin de comparer la sémantique de deux phrases et non pas la syntaxe.

3.2.1.7.4 GloVe vs Poincaré

Nous allons faire la comparaison ici d'une méthode évoluant dans un espace Euclidien (GloVe) avec une méthode évoluant dans un espace hyperbolique (Poincaré). Comme expliqué dans la rubrique « 3.2.1.6 Poincaré embeddings », Poincaré parvient à déceler une hiérarchie lors du regroupement de plusieurs termes qui sont syntaxiquement ou sémantiquement liés, par rapport à GloVe, par exemple. De plus, Poincaré parvient à réduire le nombre de dimensions que va prendre la représentation de l'ensemble des liens entre tous les mots d'un corpus comparé à un espace Euclidien dans lequel le nombre de dimensions grandit exponentiellement.(Jayant, 2017) Cependant, après plusieurs recherches, je n'ai pas pu trouver des modèles utilisant Poincaré et qui soit pré entrainé, alors que plusieurs modèles pré entrainé de GloVe existent. Aussi, plus le jeu d'entraînement est conséquent, plus les résultats ont des chances d'être concluants. De ce fait, en utilisant Poincaré, nous devrions nous même entraîner cette méthode sur des corpus de plusieurs dizaines, voire centaines de millions de mots. N'ayant pas à disposition un modèle de Poincaré déjà pré entrainé et ne disposant pas d'une puissance de calcul conséquente, il ne me sera impossible d'utiliser Poincaré et d'avoir des résultats concluants.

Les modèles prés entraînés pour Poincaré qui sont inexistants s'expliquent aussi du fait que la méthode a été publiée récemment par Facebook (2017).

Nous allons finalement opter pour la méthode GloVe concernant la phase de « Word embeddings ».

4. Siamese networks

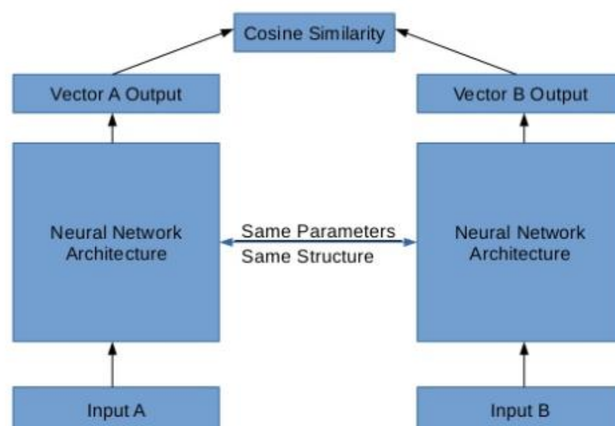
Dans le cas d'utilisation d'un réseau de neurones artificiel dans le cadre de ce travail de Bachelor, nous devons comparer deux phrases et d'ensuite déterminer si elles sont similaires. Avec un réseau de neurones de type LSTM, nous lui donnons en entrée une phrase et lui nous retourne une prédiction, or, nous aimerions lui donner deux phrases en entrée et ensuite seulement pouvoir faire une prédiction quant à leur relation sémantique. L'architecture d'un Siamese network nous permettra cela et nous allons en expliquer son fonctionnement et son origine dans la rubrique suivante.

4.1 Fonctionnement

L'architecture de Siamese network a été premièrement proposée dans le livre « Neural Networks for Fingerprint Recognition » de Pierre Baldi et Yves Chauvin en 1992. (Baldi, Chauvin, 1993) Elle a ensuite été utilisée dans plusieurs projets de recherche visant à déterminer le degré de similarité entre des images ou entre des phrases. Par exemple, un travail de recherche a été publié par Yann LeCun, un chercheur en intelligence artificielle et directeur du centre de recherche en intelligence artificielle de Facebook (LeCun, 2013), qui s'intitule « Learning Similarity Metric Discriminatively, with Application to Face Verification » et qui vise à mesurer la similarité entre deux visages pris en photo sous différents angles, par exemple. Un autre travail de recherche a également été publié par Yann LeCun et qui s'intitule « Signature Verification using a Siamese Time Delay Neural Network ». Ce travail vise à détecter les similitudes entre deux signatures écrites sur une tablette.

Un Siamese network fonctionne de telle manière qu'il est basé sur la duplication du fonctionnement d'un réseau de neurones artificiels en deux sous-réseaux. Ces deux sous-réseaux vont partager les mêmes paramètres et les mêmes poids, mais vont recevoir chacun une entrée différente et produire, par conséquent, chacun une sortie différente. L'objectif va être ensuite de comparer ces deux sorties pour pouvoir dire si ces deux mêmes entrées sont sémantiquement similaires. Voici la forme que va prendre ce réseau de manière graphique :

Figure 30 Siamese network

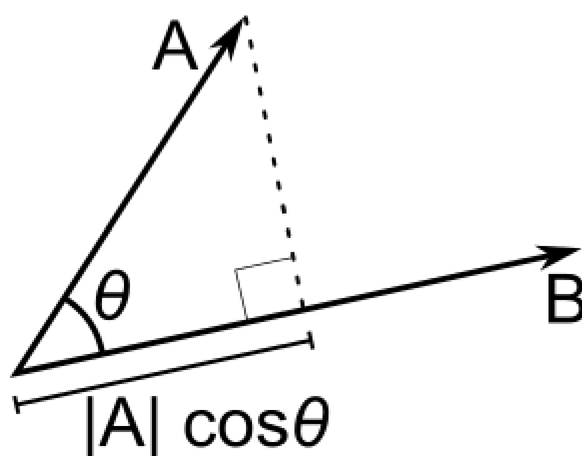


(Nicholas McClure, 2017)

Notre réseau de base travaillera donc en parallèle dans deux sous réseaux différents mais représentant la même structure, ayant les mêmes poids et les mêmes paramètres. La structure des deux sous réseaux sera celle d'un LSTM, comme celle-ci a été présentée à la section « 3. Long Short-Term Memory (LSTM) ». Il va s'agir ensuite de traiter, à chaque passage de deux entrées, le résultat retourné à l'aide de l'étape de la Figure 30 qui s'appelle « Cosine Similarity ».

Cette étape consiste à calculer la distance entre les deux vecteurs de sortie des sous réseaux. Pour calculer cette distance, nous allons ici utiliser la « Cosine distance » qui permet de placer deux vecteurs dans un espace et de calculer l'angle qui les sépare. Ainsi, nos deux vecteurs de sortie, une fois placés dans un espace, seraient représentés, par exemple, de cette manière-là :

Figure 31 Cosinus entre deux vecteurs projetés



(Perone, 2013)

Plus l'angle entre les deux vecteurs de la Figure 31 est bas et plus les deux vecteurs présentent des similarités. Prenons l'exemple où notre Siamese network nous retourne deux vecteurs $A = (2,3,1)$ et $B = (6,3,4)$ qui représentent les deux phrases à comparer en entrée. La formule à appliquer pour calculer le cosinus de ces deux vecteurs est la suivante :

$$cos\theta = \frac{\vec{A} * \vec{B}}{\|A\| * \|B\|}$$

Pour calculer le numérateur qui représente le produit scalaire entre le vecteur A et B, il s'agit simplement de faire l'étape suivante :

$$\vec{A} * \vec{B} = 2 * 6 + 3 * 3 + 1 * 4 = 25$$

Le dénominateur représente la multiplication de la taille des deux vecteurs et s'obtient de la manière suivante :

$$\|A\| = \sqrt{2^2 + 3^2 + 1^2} = \sqrt{14}$$

$$\|B\| = \sqrt{6^2 + 3^2 + 4^2} = \sqrt{61}$$

En divisant maintenant le résultat du produit scalaire par le résultat du produit de la taille des deux vecteurs, nous obtenons un cosinus de :

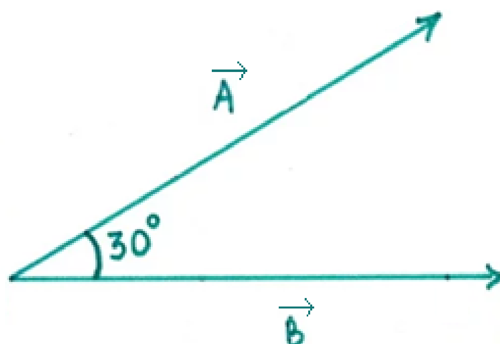
$$cos\theta = \frac{25}{\sqrt{14} * \sqrt{61}} \approx 0{,}8556$$

Une fois le cosinus obtenu, il ne nous reste plus qu'à obtenir l'angle correspondant à ce cosinus qui s'obtient en effectuant la formule suivante :

$$cos^{-1}(cos\theta) \approx 31{,}174°$$

On peut voir ici que nos deux vecteurs sont séparés d'un angle d'environ 30 degrés et qu'ils sont donc proches au niveau du sens des deux phrases qu'ils représentent, mais qu'ils ne sont pas sémantiquement équivalents à proprement dit. Dans de tels cas, nous devrons définir si nous considérons ses vecteurs suffisamment proches pour les considérer comme sémantiquement équivalents ou non. Voici un exemple graphique d'un angle de 30 degrés entre deux vecteurs qui peut vous permettre de mieux visualiser l'angle approximatif séparant le vecteur A du B :

Figure 32 Angle de 30° entre deux vecteurs



(Wikihow, 2018)

Maintenant que nous avons établi l'architecture cible, que nous savons ce qu'est un LSTM et comment nous allons calculer le degré de similarité entre deux phrases, nous allons passer au côté pratique et commencer par définir et expliquer les librairies que nous allons utiliser.

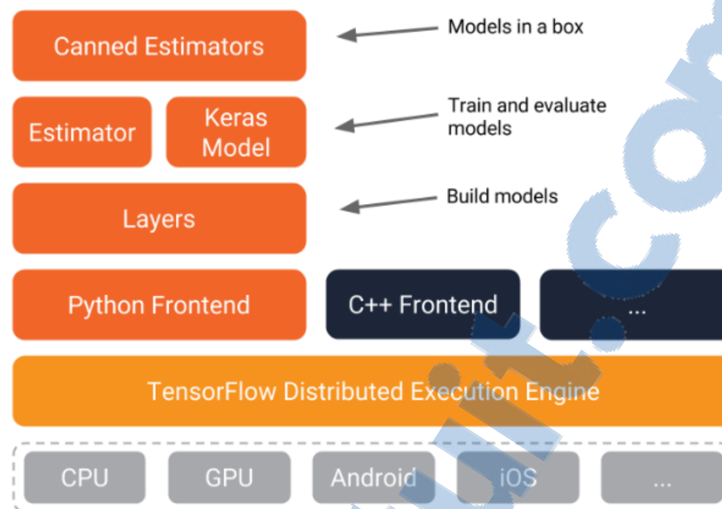
5. De la théorie à la pratique

5.1 TensorFlow

TensorFlow est une librairie de Machine Learning développée par Google en 2010 et est disponible en open-source depuis novembre 2015.(Géron, 2017) Celle-ci permet d'effectuer des calculs matriciels et vectoriels sur de larges jeux de données et elle est suffisamment générale pour permettre son utilisation sur n'importe quelle plateforme. De plus, la librairie a également été développée de sorte qu'on puisse utiliser la puissance de calcul de cartes graphiques (GPU), comme de processeurs (CPU) ou même d'un périphérique Android ou iOS.

De plus, la librairie peut être utilisée dans une multitude de langages de programmation comme le Python et le C++. Voici donc un schéma qui représente TensorFlow et la manière dont elle s'interface aux différents langages de programmation et au matériel de son utilisateur :

Figure 33 TensorFlow API



(Unruh, 2017)

Comme on peut le voir sur la « Figure 33 », la librairie dispose d'une couche appelée « TensorFlow Distributed Execution Engine » qui permet de distribuer l'exécution d'un modèle sur plusieurs types de matériel différents, voire sur plusieurs serveurs différents. Cette couche est directement liée au « frontend » qui peut être en Python, par exemple. Les différentes couches qui se trouvent ensuite en dessus de celle du « frontend » permettent de faciliter la création et l'évaluation d'un modèle de réseau de neurones. TensorFlow a donc été choisi pour sa flexibilité, mais aussi parce qu'il s'agit d'une librairie qui est largement utilisée aujourd'hui et qu'il y a, par conséquent, une communauté qui est très importante. Comme le montre la « Figure 34 » qui recense le nombre d'étoiles obtenues sur GitHub, le nombre de forks, le nombre de commits et le nombre de question posée sur le forum StackOverflow en 2017, TensorFlow se démarque et il serait donc pour moi plus facile de faire face aux problèmes que je pourrais rencontrer lors de l'élaboration de mon modèle.

Figure 34 Communauté de TensorFlow

Deep Learning Framework	Number of GitHub Stars	Number of GitHub Forks	Number of GitHub Commits Last Month	Number of Stack Overflow Questions
TensorFlow*	60030	28808	1127	12118
Caffe*	18354	11277	12	4355
Keras*	16344	5788	71	2754

(Intel.com, 2017)

Rapport-Gratuit.com

Cependant, après plusieurs recherches et l'élaboration de différents modèles de réseaux de neurones résolvant des tâches simples, je vais me tourner vers une librairie de plus haut niveau qui s'appelle Keras. Une librairie de haut niveau permet de s'interfacer avec un ou plusieurs frameworks, comme Tensorflow dans notre cas, et permet de faciliter le développement. Keras est une librairie utilisable en Python et développée par François Chollet en 2015. Elle permet à son utilisateur de pouvoir créer des modèles de réseaux de neurones en utilisant Tensorflow en arrière-plan. Par ailleurs, cette librairie permet de s'affranchir de toute la syntaxe de Tensorflow qui peut s'avérer parfois fastidieuse, d'après mon expérience personnelle, puisque Keras s'occupera de générer les bons appels de fonction en arrière-plan selon nos propres instructions.

J'ai pris cette décision dans le cadre de ce travail de Bachelor pour une raison simple qui est le manque de temps et l'objectif que je m'étais fixé au début. A savoir, de pouvoir créer et comprendre un modèle de réseau de neurones dans le cadre de l'analyse du langage naturel.

5.1.1 Les bases de Tensorflow

Comme la principale librairie utilisée sera Keras, mais qu'il s'agit véritablement de Tensorflow qui travaillera en arrière-plan, il est intéressant de comprendre les différents concepts et objets de Tensorflow.

5.1.1.1 Tensor

Un « Tensor » est un objet qui représente une généralisation d'un vecteur ou d'une matrice, il est le principal objet qui est manipulé avec Tensorflow.(Tensorflow.org, 2018) Un « Tensor » est représenté dans TensorFlow par n-dimensions et avec un type qui lui est attribué lors de son instanciation. Selon la documentation de TensorFlow, un « Tensor » a les propriétés suivantes :

- a Data Type (float32, int32, or string, for example)
- a shape

Ce qui correspond à donner un type pour toutes les valeurs contenues dans le « Tensor » et à lui donner une taille. Ce type d'objet est à la base des autres objets dans la librairie puisqu'ils vont finalement être contenus dans des « Tensor ».

5.1.1.2 Session

L'objet « Session » dans TensorFlow nous permet de déclarer un espace dans lequel la librairie allouera de la mémoire et y stockera les différentes valeurs des objets.

5.1.1.3 Variables

Il existe plusieurs types de variables différentes dans la librairie qui sont notamment :

- Variable
- Constant
- Placeholder

Dans le cas de la « variable », il s'agit de stocker l'état d'une valeur à un instant et de la préserver lors de l'exécution de l'entraînement d'un réseau de neurones. Le constructeur « Variable() » qui nous permet d'instancier une « variable » prend en paramètre la valeur de celle-ci, ainsi que son nom, qui lui est optionnel :

```
w = tf.Variable(<initial-value>, name=<optional-name>)
```

(Gülen, 2018)

La valeur ici peut être un « Tensor » et ce sera souvent le cas lors de la construction d'un modèle. Une fois la « variable » initialisée, son type et sa taille sont fixées, mais la valeur de celle-ci peut être modifiée à l'aide la méthode « assign() ». Une « variable » peut être initialisée avec une opération tel que l'exemple ci-dessous où la variable « c » est initialisée avec la valeur 3 (a+b) :

```
3 a = tf.Variable(1, tf.int32)
4 b = tf.Variable(2, tf.int32)
5 c = tf.Variable(a+b, tf.int32)
```

(Gülen, 2018)

Le deuxième type est la « constant » qui est similaire à la « variable » excepté que sa valeur d'initialisation ne peut être changée et, par conséquent, est constante.

Enfin, le troisième type est le « placeholder » qui permet de déclarer une variable dont on ne connait pas encore la ou les valeurs que celle-ci va prendre ou même la taille que celle-ci aura. Ainsi, il nous sera possible de définir sa valeur lors de l'exécution d'un modèle de réseau de neurones, par exemple. Voici un exemple de déclaration de « placeholder » qui est de type « float » et qui ne possède pas de taille prédéfinie :

```
x = tf.placeholder("float", None)
```

(Gülen, 2018)

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

On va maintenant stocker la multiplication de « x » par 2 dans une variable appelée « y ». Bien que nous ne connaissons pas encore la valeur de « x », il s'agit ici seulement de déclarer leur structure et la valeur de « y » ne sera connue que lors de l'exécution :

```
y = x * 2
```

(Gülen, 2018)

Ainsi, lorsqu'on aimerait exécuter le code pour que la valeur de « y » soit connue, il ne nous reste qu'à faire ceci :

```
with tf.Session() as session:
    result = session.run(y, feed_dict={x: [1, 2, 3]})
    print(result)
```

(Gülen, 2018)

Ici on exécute la variable « y », mais en injectant dans « x » la valeur « [1, 2, 3] ». On aura donc une valeur de « [1*2, 2*2, 3*2] » pour la variable « y » lorsque l'exécution sera finie. Pour un réseau de neurones, on pourrait donc déclarer un « placeholder » qui représente les phrases que nous allons lui donner en entrée et les données ne lui seront passées que pendant l'exécution de celui-ci.

5.1.2 Graphes computationnels

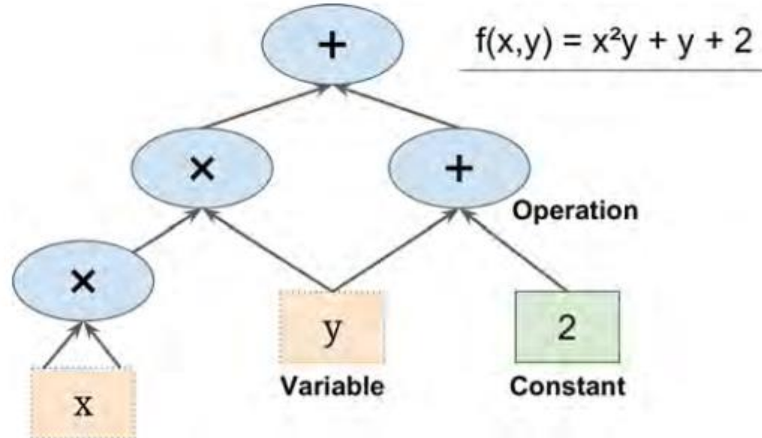
Nous avons démontré dans la section « 5.1.1 Les bases » comment déclarer une ou plusieurs variables et comment effectuer une opération sur celles-ci. Nous allons ici expliquer comment la déclaration est interprétée par la librairie TensorFlow. A chaque fois que nous déclarons une variable, la librairie va ajouter celle-ci dans un arbre computationnel, appelé aussi graphe computationnel. Aussi, dès que nous déclarerons une opération entre deux variables, TensorFlow créera un nœud qui représentera notre opération et duquel partiront les deux variables. Ainsi, lors de l'exécution du code, la librairie n'aura plus qu'à utiliser le graphe déclaré et l'exécuter en C++ optimisé.(Géron, 2017) Voici un exemple d'équation que nous pourrions déclarer :

Équation 2 Exemple équation pour un graphe computationnel

$$f(x, y) = x^2y + y + 2$$

Et voici comment TensorFlow l'interprète avec un graphe computationnel (ce graphe n'est pas celui généré par la librairie mais sa représentation pour qu'elle soit plus lisible) :

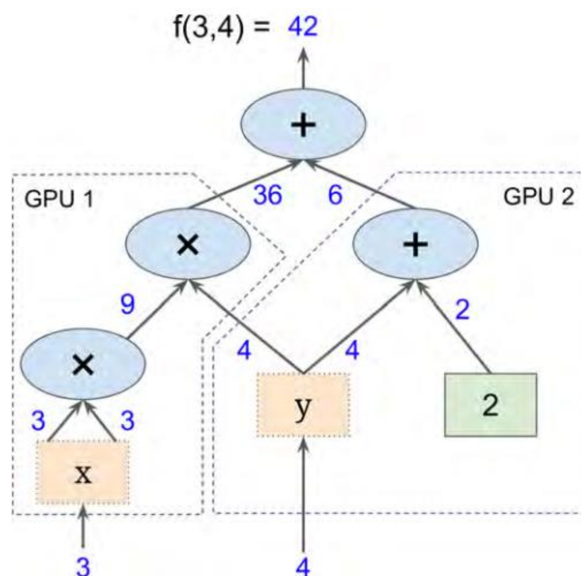
Figure 35 Représentation d'un simple graphe computationnel



(Géron, 2017)

L'avantage est que nous pourrions séparer en deux un tel graphe pour pouvoir exécuter une partie des calculs sur une carte graphique et l'autre partie sur une autre carte graphique, ce qui nous permettrait de calculer de manière parallèle et de gagner en temps d'exécution. Voici une figure qui démontre les valeurs calculées par TensorFlow lors de l'exécution du graphe à la « Figure 35 » sur deux cartes graphiques :

Figure 36 Représentation d'un graphe computationnel multi-GPU



(Géron, 2017)

On déclare ici le « y » étant égal à « 4 » et le « x » étant égal à « 3 ». La partie de gauche s'exécute sur une carte graphique notée « GPU 1 » et la partie de droite sur une autre notée « GPU 2 ». Une fois l'exécution de ce graphe lancée, TensorFlow lancera l'exécution des opérations en partant du bas et en remontant jusqu'à la racine. C'est le moteur d'exécution qui s'occupe de lancer une partie des opérations sur l'une ou l'autre des cartes graphiques et d'en récupérer le résultat pour la dernière opération. Enfin, voici un exemple de code que nous aurions écrit pour que la librairie crée le graphe cité à la « Figure 35 » :

```
14 x = tf.Variable(2, name="x")
15 y = tf.Variable(4, name="y")
16 f = x*x*y + y + 2
```

(Gülen, 2018)

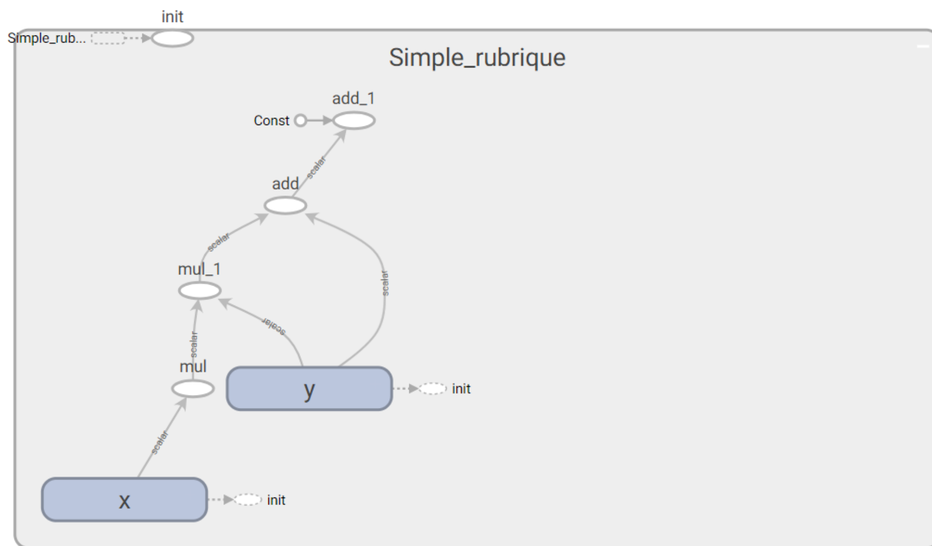
Outre le fait que les graphes nous permettent de paralléliser certains calculs, ils sont utiles pour visualiser de manière globale l'ensemble des opérations d'un modèle. En revanche, si le modèle devient assez complexe, il peut être fastidieux et difficile de le comprendre.

5.1.2.1 Scope

Un « Scope » est une manière de regrouper un ensemble d'opérations dans une seule et même rubrique lorsque TensorFlow crée le graphe computationnel. Ceci nous permet d'améliorer la lisibilité du graphe construit et de mieux le comprendre. En effet, lorsqu'il s'agit de faire un réseau de neurones plutôt qu'une simple addition et multiplication, cela devient vite illisible pour un humain.

Pour avoir un exemple de graphe, nous allons utiliser Tensorboard qui est un outil fourni avec TensorFlow et qui nous permet de visualiser des graphes computationnels ou des graphiques nous permettant d'analyser la performance d'un modèle. Voici un exemple de graphe lors de la déclaration de « l'Équation 2 » avec TensorFlow:

Figure 37 Exemple de graphe avec Tensorboard



Sur la « Figure 37 » on peut apercevoir plusieurs objets représentés par des formes différentes. Voici la correspondance de chacun de ceux- ci :

Figure 38 : Définition des symboles d'un graphe computationel

(Gülen, 2018)

Ce graphe se lit de bas en haut, exactement comme la « Figure 35 ». Si une seule variable est reliée à une opération, cela signifie que les deux termes de l'opération sont représentés par la même variable. Dans la « Figure 37 », on commence donc par multiplier la variable « x » avec elle-même (opération « mul »), on multiplie ensuite le résultat avec la variable « y » (opération « mul_1 »). On va ensuite additionner la variable « y » au résultat de l'opération « mul_1 » (opération « add »). Enfin, on va additionner une constante au résultat final. On peut donc voir que le résultat de ce graphe déclare exactement « l'Équation 2 » et le code pour que TensorFlow génère ce graphe a été celui-ci :

```
14 import tensorflow as tf
15
16 tf.reset_default_graph()
17
18 with tf.name_scope('Simple_rubrique') as scope:
19     x = tf.Variable(2, name="x")
20     y = tf.Variable(4, name="y")
21     z = tf.constant(2)
22     f = x*x*y + y + z
23
24 init = tf.global_variables_initializer()
25 sess = tf.Session()
26
27 sess.run(init)
28
29 train_writer = tf.summary.FileWriter( 'C:/tmp/TensorFlow_graphs',
30                                       sess.graph)
```

(Gülen, 2018)

On doit tout d'abord remettre par défaut les valeurs de notre graphe à la ligne 16 pour éviter de superposer plusieurs déclarations des mêmes variables, par exemple. On déclare ensuite un « Scope » à la ligne 18 pour regrouper l'ensemble des variables et opérations de notre exemple. On va ensuite initialiser les variables, lancer notre session et enregistrer le graphe pour pouvoir le visualiser à l'aide de Tensorboard. Tensorboard est un outil fourni avec TensorFlow et nous permet de visualiser des graphes computationnels ou des graphiques nous permettant d'analyser la performance d'un modèle.

5.2 Phase d'apprentissage

Maintenant que nous avons vu les bases de la librairie TensorFlow et comment elle fonctionne, nous allons aborder le sujet de la phase d'apprentissage. En effet, lors de la création d'un réseau de neurones, celui-ci est initialisé avec des poids qui sont aléatoires et ses résultats le sont donc aussi. Le défi réside dans l'apprentissage que ce modèle effectuera afin qu'il soit de plus en plus performant.

Cette phase est celle qui prend le plus de temps puisqu'il s'agira d'effectuer différents tests jusqu'à trouver l'architecture qui offre les meilleures précisions. Il n'existe pas de modèle qui résolve tout type de problèmes et par conséquent il est nécessaire de réaliser plusieurs essais. Cependant, il existe des bonnes pratiques que nous suivrons et qui garantiront une base rigoureuse pour chacune des architectures que nous allons construire.

5.2.1 Bonnes pratiques

5.2.1.1 Général

Commençons cette section de bonnes pratiques par aborder de la proportion du jeu d'entraînement par rapport à toutes les données que nous avons à disposition. Celui-ci ne doit être utilisé que pour l'entraînement et avoir une certaine proportion. En effet, nous allons d'abord passer par une phase d'entraînement, puis une phase d'évaluation et enfin une phase de test. Il est donc important d'utiliser une partie différente du même jeu de données pour chaque étape. Aussi, si la partie du jeu de données utilisée pour l'entraînement l'est également pour l'évaluation et le test, les résultats risquent d'être bons, mais de manière trompeuse. C'est-à-dire que sur ce jeu de données les résultats sont bons mais ne le seraient pas ou beaucoup moins sur un autre.

Le but de séparer nos données résulte dans le fait que notre réseau commencera par apprendre pendant le premier cycle d'entraînement, aussi appelé « epoch », et que nous allons l'évaluer avec le jeu d'évaluation à chaque fois que celui-ci aura été entraîné sur tout le jeu d'entraînement. Ainsi, il pourra adapter les différents poids du réseau, à l'aide de l'algorithme du gradient, et s'améliorer pour « l'epoch » suivant. Il est impératif que le jeu d'évaluation et de test soient des données différentes du jeu d'entrainement pour éviter que notre réseau apprenne toutes les données et ne sache faire des prédictions satisfaisantes que sur ce jeu de données.

Concernant la taille du jeu d'entraînement, il n'y a pas de règle qui la précise clairement. Cependant, il existe une bonne pratique qui est soutenue par la documentation en Machine Learning d'Amazon qui est de séparer le jeu de données avec un ratio de 70-15-15. C'est-à-dire que nous aurons un jeu d'entraînement composé de 70% de notre jeu de données, un jeu de test composé de 15% et un jeu de validation composé des 15% restants. (Amazon, 2018)

Il est ensuite primordial d'avoir un jeu de données suffisamment important pour éviter de faire de « l'overfitting ». C'est-à-dire d'éviter de surentrainer notre modèle pour qu'il puisse être généralisable envers des données qu'il n'aurait jamais vues. En effet, si nous ne disposons que de très peu de données, il nous faudra passer régulièrement les mêmes données à notre réseau et il sera en mesure d'apprendre comment prendre des décisions spécifiquement à ces données-là, mais pas sur de nouvelles données. Et si nous disposons que de très peu de données et que nous voulons éviter de faire de « l'overfitting » en passant nos données peu de fois dans le réseau, celui-ci risque de faire de « l'underfitting ». C'est-à-dire qu'il ne va pas traiter assez de données et qu'il ne pourra pas fournir des taux de précision satisfaisants dans ses prédictions et

risquera d'être sous-entraîné.(Brownlee, 2016b) Encore une fois, il n'y a pas de règle qui précise la taille du jeu de données minimale, mais beaucoup affirment que plus la tâche est complexe, plus il faut de données pour que le modèle soit correctement entraîné. (Brownlee, 2017b)

5.2.1.2 Hyperparamètres

Les « hyperparameters » sont des paramètres qui vont être appliqués sur notre modèle de réseau de neurones et peuvent significativement agir sur la précision de celui-ci. Nous allons donc aborder ceux pour lesquelles certaines règles de bonnes pratiques existent.

Le premier paramètre est le nombre de couches cachées de notre réseau de neurones. Il est recommandé de commencer par avoir une seule couche cachée et ensuite d'essayer d'en rajouter selon les résultats obtenus.(Géron, 2017) En effet, la plupart des problèmes de classification peuvent être résolus à l'aide d'une seule couche cachée, mais suivant la complexité de celui-ci, il conviendra peut-être mieux d'en avoir plusieurs pour avoir des résultats satisfaisants.(Géron, 2017)

Le deuxième paramètre est le nombre de neurones par couche cachée. Le nombre de neurones de la couche d'entrée et de sortie vont dépendre de la structure d'entrée de nos données et de la structure de fin attendue. Pour la couche cachée, il devient courant d'utiliser une approche dite de « funnel » où le nombre de neurones va diminuer en avançant dans les couches cachées.(Géron, 2017) Cependant, il n'existe pas de recommandation précise si notre réseau ne contient qu'une seule couche cachée, hormis le fait qu'il faut augmenter graduellement le nombre de neurones jusqu'au moment où celui-ci commence à faire de l'overfitting.(Géron, 2017) Ceci, pourrait donc être un indicateur du nombre de neurones à ne pas dépasser pour une couche cachée.

Le troisième paramètre est le choix de la fonction d'activation. Il est conseillé d'utiliser la fonction « softmax » pour la sortie finale de notre réseau si nous faisons de la classification.(Géron, 2017) Celle-ci ne nous retourne que des valeurs entre 0 et 1 et le total de toutes les sorties fournies, pour une instance, sont égal à 1. Les résultats de cette fonction s'apparentent donc à des pourcentages et pour un exemple de modèle qui nous dit si sur une photo il s'agit d'un homme ou d'une femme, nous obtiendrions par exemple :

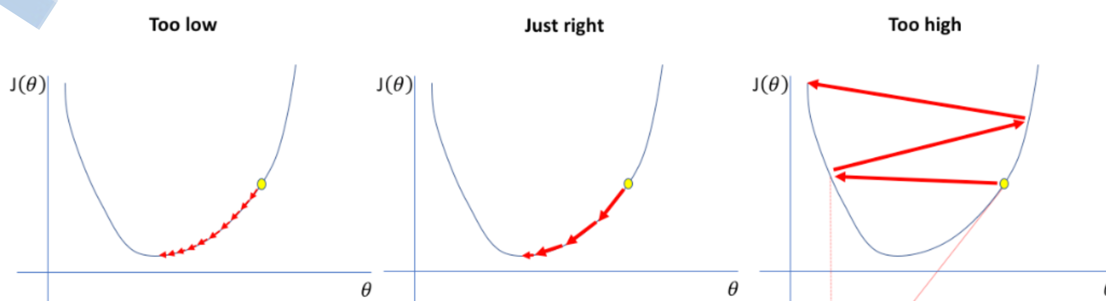
- 16 % de probabilité que l'instance soit une femme
- 84% de probabilité que l'instance soit un homme

Concernant la fonction d'activation des neurones des ou de la couche cachée, il n'existe pas de recommandation particulière pour les réseaux de neurones récurrents.

Le quatrième paramètre est le « Dropout » qui est une technique de régularisation proposée par Srivastava en 2014.(Srivastava et al., 2014) Celui-ci permet d'éviter un maximum le phénomène « d'overfitting » en sélectionnant aléatoirement certains neurones qui seront ignorés durant un cycle d'entraînement. Ceci permet d'éviter que les neurones se « spécialisent » dans une tâche et permet donc de généraliser un maximum le modèle pour que même si un ou plusieurs neurones sont temporairement désactivés, il puisse donner des prédictions satisfaisantes. Ce paramètre s'exprime en un ratio allant de 0 à 1. Plus il est élevé, plus il y aura de neurones désactivés temporairement et il est conseillé d'avoir un dropout entre 0.2 et 0.5 pour commencer, car un dropout trop élevé entraînerait le phénomène « d'underfitting ». (Brownlee, 2016a)

Le cinquième paramètre est le « Learning rate ». Celui-ci va contrôler l'ampleur avec laquelle le réseau va adapter ses poids pour s'améliorer dans ses prédictions et varie généralement entre 1 et des petites valeurs telles que 0.0001. Plus il est élevé, plus les poids vont être affectés pas de grands changements et inversement s'il est petit. Le but en changeant les poids dans le réseau est de diminuer le résultat de la fonction de coût (la différence entre le résultat attendu et celui obtenu). Ce qui fait que si le « Learning rate » est trop élevé, l'adaptation des poids sera toujours trop élevée pour atteindre une valeur assez précise pour avoir un résultat satisfaisant et on risque de ne jamais réussir à trouver le minimum global de la fonction de coût. Inversement, si le Learning rate est trop faible, on aura besoin d'un très grand nombre d'entraînements pour arriver au minimum global. Voici un schéma qui illustre les différents cas de figure :

Figure 39 Fonction de coût et Learning rate



Rapport-Gratuit.com

Une recommandation est de commencer avec un Learning rate élevé, disons 0.01, et de le diminuer après quelques epoch, disons à 0.001, afin d'atteindre un minimum global aussi vite que possible.(Despois, 2016)

5.3 Phase de test

5.3.1 Bonnes pratiques

Dans cette section nous allons discuter de la manière dont nous allons tester notre modèle. A la rubrique précédente, nous avons parlé plus en détail du jeu d'entraînement et nous allons, ici, nous intéresser au jeu d'évaluation et à celui de test.

Le jeu d'évaluation est celui qui est utilisé pour tester notre modèle à chaque fin de cycle d'entraînement. C'est-à-dire qu'une fois que le jeu d'entraînement a été passé à notre modèle et que les prédictions ont été émises, nous donnons à notre modèle le jeu d'évaluation. Comme le modèle est évalué avec des données sur lesquelles il ne s'est jamais entraîné, cela simulera l'arrivée de nouvelles données qui sont inconnues et nous permettra donc de savoir si le modèle est généralisable. Plus le modèle est général, plus il pourra traiter des données inconnues avec précision.

Le jeu de test est celui qui est utilisé lors de la phase de test finale du réseau de neurones. Une fois que le nombre « d'epoch » que nous avons décidé d'effectuer est terminé et que le modèle a été évalué à chaque epoch, nous devons lui fournir un nouveau jeu de données sur lequel on va pouvoir le tester.

Enfin, comme expliqué dans la section « 5.2.1 Bonnes pratiques », la taille du jeu d'évaluation et du jeu de test sera de 30% du jeu de données total. Chacun des deux jeux de données aura donc une taille de respectivement 15 %.

6. Création du modèle

Nous avons désormais tous les outils nécessaires à la réalisation de notre réseau de neurones récurrent à savoir un Siamese LSTM. Commençons par discuter du jeu de données et ensuite nous établirons toutes les différentes étapes qui nous permettront de construire petit à petit notre architecture.

6.1 Jeu de données

Pour entraîner notre futur réseau de neurones, nous avons besoin d'un fichier qui contient des paires de phrases et une annotation pour chaque paire qui affirme si ces

deux phrases son sémantiquement similaires. Ainsi, nous pourrons informer notre réseau, lors de la phase d'entraînement, si ses prédictions sont correctes ou non.

Après de nombreuses recherches, je suis enfin tombé sur un jeu de données conséquent et annoté. Celui-ci a été créé par Quora qui est une entreprise Web co-fondée en 2009 par Adam D'Angelo et Charlie Cheever. Quora propose un service où tout un chacun peut venir poser une question et en récolter les réponses d'autres utilisateurs, tout ceci organisé par sujet.(Robbins, 2017) Le jeu de données mis à disposition par Quora contient plus de 400'000 paires de phrases anglaises en open source avec à chaque fois un attribut qui rend la valeur de 1 si les deux phrases sont similaires et 0 dans le cas contraire. Le jeu de données se présente ainsi :

Figure 40 Quora paires de questions

7	Astrology: I am a Capricorn Sun Cap moon and cap rising...what does that say about me?	I'm a triple Capricorn (Sun, Moon and ascendant in Capricorn) What does this say about me?	1
8	Should I buy tiago?	What keeps childern active and far from phone and video games?	0
9	How can I be a good geologist?	What should I do to be a great geologist?	1
10	When do you use シ instead of し?	When do you use "&" instead of "and"?	0

(Shankar et al., 2017)

Pour la première construction de notre modèle, nous allons utiliser les 50'000 premières paires de phrases du jeu de données de Quora pour éviter d'avoir un temps d'entraînement trop long. Néanmoins, nous pourrons augmenter ce nombre une fois que l'architecture sera pleinement fonctionnelle et optimisée.

6.2 API Embeddings

Maintenant que nous avons un jeu de données à notre disposition, nous allons construire notre modèle petit à petit.

La première étape est de trouver un moyen de transformer nos phrases en entrée en nombres entiers ou réels que le réseau de neurones va pouvoir utiliser dans ses fonctions de coût. Comme expliqué à la section « 3.2.1 Word embeddings », nous allons utiliser GloVe pour générer des représentations numériques de chacun des mots de chacune de nos phrases. Cependant, nous n'allons pas directement utiliser l'algorithme, mais plutôt le résultat de l'utilisation de cet algorithme sur plusieurs

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

millions de mots. En effet, nous allons utiliser un fichier texte contenant 840 millions de mots et caractères différents avec pour chacun un vecteur d'une taille de 300 qui le représente. Ce vecteur contient uniquement des nombres réels négatifs ou positifs et ceux-ci ont été généré à l'aide de l'utilisation de l'algorithme de GloVe et voici un exemple de quelques valeurs du vecteur représentant le mot « the » :

```
the -0.038194 -0.24487 0.72812 -0.39961 0.083172 0.043953  …
```

Ce fichier texte a été mis à disposition par le Common Crawl qui est une organisation web à but non lucratif fondée en 2007 et qui fournit différentes ressources en open-source. (Commoncrawl.org, 2007)

Désormais, nous avons besoin de parcourir le fichier texte cité plus haut pour chacun des mots et caractères de chacune des phrases et en extraire sa représentation. Le problème étant que le fichier texte pèse plus de 5 Gb, ce qui fait que si nous devons le parcourir depuis le début jusqu'à avoir trouvé la représentation du mot recherché, ceci prendrait énormément de temps. J'ai préféré donc créer une API qui charge l'ensemble du fichier dans la mémoire RAM et le garde tant que celle-ci n'est pas stoppée manuellement. D'une part, le temps d'accès au fichier en question est beaucoup plus rapide. D'autres parts, si je transforme l'ensemble des phrases par le biais d'une API, notre modèle de réseau de neurones n'aura pas besoin de contenir une couche de « word embedding ». En effet, les données fournies en input du réseau seront donc déjà prêtes à être traitées. Il s'agit d'un avantage dans notre cas d'utilisation, car la couche de « word embedding » aurait dû être initialisée avec l'ensemble des représentations de nombres réels de chacun des mots uniques rencontrés dans le jeu de données sous la forme d'une matrice. Ainsi, lors de la phase de test, nous aurions eu exactement la même matrice. De ce fait, si des mots n'ayant jamais été rencontrés dans le jeu d'entraînement se retrouvaient dans le jeu de test, ils n'auraient pas de représentation dans cette matrice et serait par défaut ignorés avec l'affectation d'un vecteur ne contenant que des « 0 ».

Pour en revenir à l'API, on commence par importer la librairie flask qui nous permet de créer une API python requêtable. Ensuite, nous initialisons l'API à la ligne 16, puis à la ligne 18 nous permettons à tous types de navigateur d'envoyer une requête HTTP à cette API et enfin à la ligne 19 nous activons le mode « debug » pour permettre de retrouver plus facilement l'origine de problèmes qui pourraient survenir.

```
15 import flask
16 app = flask.Flask(__name__)
17 from flask_cors import CORS
18 CORS(app, resources={r"/*": {"origins": "*"}})
19 app.debug = True
```

(Gülen, 2018)

Avec le code suivant, nous allons créer une méthode que nous allons pouvoir appeler avec une requête HTTP de type « Post » et avec la route suivante « http://127.0.0.1/tense_embedding » :

```
22 @app.route("/tense_embedding", methods=["POST"])
23 def tense_embedding () :
```

(Gülen, 2018)

Cette méthode se chargera de recevoir un objet JSON - JSON est une syntaxe permettant le transport de données (json.org, 2018) – qui contiendra un attribut « tense » et qui référencera la phrase à transformer. Celle-ci s'attend à recevoir une phrase avec un espace entre chaque mot et va donc créer une liste contenant à chaque indice un mot de la phrase reçue. Elle va ensuite pour chaque mot tenter de trouver une représentation de nombres réels dans le fichier texte citer plus haut. Dans le cas où le mot n'est pas référencé dans le fichier, nous lui attribuons un vecteur rempli de zéros, ce qui équivaut à l'ignorer. Enfin, comme la librairie flask ne permet pas de retourner de tableaux d'objets, j'ai été obligé de retourner toutes les représentations de mots pour une phrase dans un seul attribut de type String, qui se nomme « embedding_string », en les séparant par un caractère que j'ai choisi et qui est un « # ». Voici le code de la méthode :

```
24      data = {"success": False}
25      json_Object = flask.request.get_json()
26      tense = json_Object['tense']
27      tense_array = tense.split()
28      string_embedding_vector = ""
29      #We iterate through the tense and get the embedding vector for each word
30      for i in range(len(tense_array)) :
31          try:
32              res = embedding_model[tense_array[i]]
33          except (Exception) :
34               res = np.full(300, 0)
35          #As we could just return a string with flask library, we concatenate all vectors in one string
36          #Separated by the "#" character
37          string_embedding_vector= string_embedding_vector+str(res)+" # "
38      data["success"] = True
39      data["embedding_string"] = str(string_embedding_vector)
40      print("tense embedding ok")
41      return flask.jsonify(data)
```

(Gülen, 2018)

Pour que cette API soit prête à être requêtée, il faut encore charger le fichier texte dans la mémoire RAM à chaque fois que l'on lance l'API. Pour cela j'ai utilisé la librairie « gensim » qui me permet de charger des modèles binaires de type « word2vec » cités

dans la section « 3.2.1.3 Word2Vec ». Cependant, nous utilisons un modèle de type texte de GloVe. Heureusement, grâce à ces quelques lignes de code, nous pouvons transformer notre fichier texte en fichier binaire avec le même format que celui de « word2vec » :

```
1 from gensim.test.utils import get_tmpfile
2 from gensim.models import KeyedVectors
3 import os
4
5 glove_file = os.getcwd() + './glove.840B.300d.txt'
6 tmp_file = get_tmpfile("glove_word2vec_format.txt")
7 from gensim.scripts.glove2word2vec import glove2word2vec
8 glove2word2vec(glove_file, tmp_file)
9 model = KeyedVectors.load_word2vec_format(tmp_file)
10 model.save_word2vec_format("glove_word2vec_format.bin", binary=True)
11
12 del tmp_file
```

(Gülen, 2018)

Grâce à la ligne 10, nous sauvegardons la transformation pour ne devoir l'exécuter qu'une seule fois étant donné que cela prend une dizaine de minutes. Enfin, nous définissons le chemin du fichier binaire généré et le chargeons à l'aide de la librairie gensim, puis nous lançons l'exécution de l'API qui peut désormais être requêtée :

```
43 model_path = './glove_word2vec_format.bin'
44 embedding_model = gensim.models.KeyedVectors.load_word2vec_format(model_path, binary=True)
45 print("app run")
46 app.run(host='127.0.0.1', port='5001')
```

(Gülen, 2018)

6.3 Entraînement du modèle

Une fois que le jeu de données, cité à la section « 6.1 Jeu de données », est téléchargé, nous allons commencer par le charger dans la mémoire, à l'aide de la librairie python « pandas » pour pouvoir l'utiliser par la suite :

```
import pandas as pd
train = pd.read_csv('train.csv')
```

(Gülen, 2018)

Comme la totalité des paires de question est désormais stockée dans la variable « train » qui est structurée comme une matrice d'objets, nous allons transformer la totalité des questions n°1 de chaque paire en une liste d'objets de type « String » - le type « String » représente les objets de type texte en programmation - pour pouvoir

effectuer les traitements suivants et nous ferons la même chose pour les questions n°2. Voici les lignes de code effectuées :

```
12 q1 = pd.Series(train.question1.tolist()).astype(str)
13 q2 = pd.Series(train.question2.tolist()).astype(str)
```

(Gülen, 2018)

Nous allons ensuite définir une longueur maximum pour chacune des phrases pour qu'ensuite on puisse restocker toutes les paires dans une seule et même matrice. Une matrice ne peut accepter que des objets de taille similaire et cette structure nous simplifiera la tâche lors de l'entraînement du modèle en nous permettant de donner directement la matrice en entrée du réseau de neurones et de ne pas devoir lui fournir les entrées une à une.

```
14 max_seq_length = 50
```

(Gülen, 2018)

Ici nous allons parcourir la liste de questions n°1 des paires de phrases et, à chacune d'entre elles, nous allons envoyer une requête HTTP de type « Post » à notre API que nous avons abordé à la section « 6.2 API Embeddings ». Pour ceci, nous utiliserons la librairie « httplib2 » qui permet de faire des requêtes HTTP de manière très rapide et avec tout un panel de paramètres à notre disposition :

```
16 from httplib2 import Http
17 for i in range(len(q1)) :
18     http_obj = Http()
19     resp, content = http_obj.request(
20         uri='http://127.0.0.1:5001/tense_embedding',
21         method='POST',
22         headers={'Content-Type': 'application/json; charset=UTF-8'},
23         body=dumps({'tense': q1[i]}),
24     )
```

(Gülen, 2018)

Une fois la réponse de la requête récupérée, des tests sont effectués afin de d'établir si le processus s'est bien déroulé. Dans ce cas, nous commençons par décoder la phrase envoyée en String dont les guillemets ont été remplacés par des apostrophes par la librairie flask. Ainsi, nous pouvons transformer ce String en objet JSON et accéder à l'attribut « embeding_string » qui contient la phrase transformée par l'API. Comme chacun des mots de la phrase sont désormais des vecteurs d'une taille de 300, mais dans un objet de type String, il faut supprimer les caractères «] » et « [», aux lignes 28-29-30, pour n'avoir plus que les « # » qui séparent chacun des vecteurs.

Ainsi, nous pouvons désormais utiliser la méthode « split » qui permet de séparer un String en un tableau de String selon un caractère. Nous supprimons à la ligne 32 le dernier élément du tableau qui se trouvait être un espace et qui ne nous intéresse pas et nous créons un vecteur nommé « question_prepared », en deux dimensions, rempli de zéros et de taille 50 * 300 (longueur de chaque vecteur de chaque mot). Ainsi, si la phrase transformée est plus courte que la longueur maximum, les derniers indices du vecteur ne contiendront que des vecteurs de zéros pour qu'ils aient tous la même taille.

```
25     if resp.status == 200 :
26         my_json = content.decode('utf8').replace("'", '"')
27         data = json.loads(my_json)
28         question_embedding_string = data['embedding_string'].replace("[ ", "")
29         question_embedding_string = question_embedding_string.replace("[", "")
30         question_embedding_string = question_embedding_string.replace("]","")
31         question_embedding_array = np.array(question_embedding_string.split(' # '))
32         question_embedding_array = question_embedding_array[0:len(question_embedding_array)-1]
33         question_prepared = np.zeros((max_seq_length,300))
34         end= max_seq_length
```

(Gülen, 2018)

Nous devons maintenant encore parcourir le tableau rempli de vecteur mais sous forme de String. Pour cela, nous devons d'abord tester si la longueur du tableau est plus petite ou plus grande que la longueur maximum pour que l'indice de parcours du tableau soit au maximum de 50. Ensuite, à chaque index du tableau nous avons une liste de nombre réels séparés par des espaces et sous forme de String tel que cet extrait :

```
-0.14409   -0.34428   0.87238   0.9225   -0.36545   -0.6996
```

Nous devons donc utiliser à nouveau la méthode « split » pour transformer ce String en un tableau de String et pour ensuite changer le vecteur de zéros à l'indice correspondant du vecteur « question_prepared » par celui-ci. Une fois la phrase complètement parcourue, nous remplaçons son ancienne valeur par la valeur de « question_prepared » qui est, pour rappel, un vecteur qui contient un vecteur de nombre réels à chacun de ses indices. Si un problème survient, le script s'arrête avec la ligne 41 et enfin nous testons l'indice « i » de parcours de la liste des question n°1 et nous l'arrêtons après avoir transformer 70'000 phrases.

```
35         if max_seq_length > len(question_embedding_array) :
36             end = len(question_embedding_array)
37         for j in range(end) :
38             question_prepared[j] = question_embedding_array[j].split()
39         q1[i] = question_prepared
40     else :
41         break
42     if i == 70000 :
43         print("Embedding q1 finished.")
44         break
```

(Gülen, 2018)

Ceci est le maximum que je puisse effectuer avec les 24 Giga de mémoire RAM que j'ai sur mon PC. En effet, à chaque fois que nous stockons des valeurs dans une variable, celle-ci est en fait stockée dans la mémoire RAM et donc plus une matrice ou un vecteur est grand, plus il prend de place. Par exemple, si nous assumons qu'une cellule de matrice prend 1 byte d'espace mémoire et que nous voulons traiter l'ensemble des 440'000 paires de phrases, soit 880'000 phrases, il faudrait déjà 880'000 bytes de mémoire pour stocker chaque phrase dans une seule cellule. Mais dans notre cas, chaque phrase a une taille de 50, ce qui fait que notre matrice contient 880'000 vecteurs d'une taille de 50, soit 44'000'000 de bytes. En outre, une fois chacun des mots transformés en valeurs réelles, ceux-ci sont représentés par un vecteur d'une taille de 300, ce qui fait que notre matrice aurait une taille finale de 880'000 * 50 * 300 = 13'200'000'000 bytes. Comme 1'000'000'000 bytes équivalent à 1GB, nous aurions besoin de plus de 13 Giga de mémoire RAM uniquement pour cette matrice, en plus des 5,5 GB du fichier GloVe déjà en mémoire (cf. section « 6.2 API Embeddings »). (gbmb.org, 2018)

Une fois le traitement terminé pour la liste des question n°1 de chaque paire, nous faisons exactement la même chose mais pour la liste de question n°2 :

```
46 for i in range(len(q2)) :
47     http_obj = Http()
48     resp, content = http_obj.request(
49         uri='http://127.0.0.1:5001/tense_embedding',
50         method='POST',
51         headers={'Content-Type': 'application/json; charset=UTF-8'},
52         body=dumps({'tense': q2[i]}),
53     )
54     if resp.status == 200 :
55         my_json = content.decode('utf8').replace("'", '"')
56         data = json.loads(my_json)
57         question_embedding_string = data['embedding_string'].replace("[ ", "")
58         question_embedding_string = question_embedding_string.replace("[", "")
59         question_embedding_string = question_embedding_string.replace("]","")
60         question_embedding_array = np.array(question_embedding_string.split(' # '))
61         question_embedding_array = question_embedding_array[0:len(question_embedding_array)-1]
62         question_prepared = np.zeros((max_seq_length,300))
63         end= max_seq_length
64         if max_seq_length > len(question_embedding_array) :
65             end = len(question_embedding_array)
66         for j in range(end) :
67             question_prepared[j] = question_embedding_array[j].split()
68         q2[i] = question_prepared
69     else :
70         break
71     if i == 70000 :
72         print("Embedding q2 finished.")
73         break
```

(Gülen, 2018)

Nous pouvons maintenant assembler les deux listes des 70'000 premières paires de phrases désormais transformées, stocker les 70'000 premières valeurs à prédire pour ces paires et supprimer la matrice contenant les 440'000 paires.

```
75 #Put all the transformed tenses in the same structure and get the 70'000 first y_lbael
76 qs = pd.DataFrame({'q1': q1[:70000], 'q2': q2[:70000]})
77 questions_cols = ['q1', 'q2']
78 X = qs[questions_cols]
79 y = train['is_duplicate'][:70000]
80 del train
```

(Gülen, 2018)

A présent, nous allons séparer notre jeu de données en un jeu d'entraînement contenant 85% des 70'000 paires et un jeu de validation contenant le reste à l'aide de la librairie sklearn.

Nous pouvons également supprimer la variable « X » et « qs » dont nous n'avons plus besoin.

```
82 from sklearn.model_selection import train_test_split
83 X_train, X_val, y_train, y_val = train_test_split(X, y, test_size = 0.15, random_state = 102)
84
85 del X
86 del qs
```

(Gülen, 2018)

Pour plus d'aisance, nous stockons les questions n°1 et n°2 du jeu d'entraînement et de validation dans des matrices ayant pour nom de colonne « left » représentant les questions n°1 et « right » représentant les questions n°2. La colonne left correspondra au côté gauche de notre Siamese network et l'autre au côté droit.

```
88 # Preparing train and validation feature-sets
89 X_train = {'left': X_train.q1, 'right': X_train.q2}
90 X_val = {'left': X_val.q1, 'right': X_val.q2}
91
92 # Preparing target labels
93 y_train = y_train.values
94 y_val = y_val.values
```

(Gülen, 2018)

Comme l'utilisation de la méthode « train_test_split » de la librairie « sklearn » nous retourne un des tableaux en deux dimensions et que nous aurons besoin d'un Tensor en 3 dimensions, nous créons deux vecteurs en 3 dimensions remplis de zéros avec le nombre de paires de phrases comme premier axe, la taille de chacune des phrases en seconde axe et avec la dimension de chaque représentation de mot comme troisième axe.

Nous parcourons ensuite le jeu de validation et le jeu d'entraînement en attribuant chaque vecteur de phrase à un indice de notre tableau en 3 dimensions.

```
100 X_train_left = np.zeros((len(X_train['left']),max_seq_length,300))
101 X_train_right = np.zeros((len(X_train['right']),max_seq_length,300))
102 index=0
103 for i, v in X_train['left'].items():
104     X_train_left[index] = v
105     index= index+1
106 index=0
107 for i, v in X_train['right'].items():
108     X_train_right[index] = v
109     index= index+1
110
111 X_val_left = np.zeros((len(X_val['left']),max_seq_length,300))
112 X_val_right = np.zeros((len(X_val['right']),max_seq_length,300))
113 index=0
114 for i, v in X_val['left'].items():
115     X_val_left[index] = v
116     index= index+1
117 index=0
118 for i, v in X_val['right'].items():
119     X_val_right[index] = v
120     index= index+1
```

(Gülen, 2018)

Les inputs étant maintenant près à être passer à notre réseau de neurones, nous pouvons construire le modèle. Commençons par définir quelques constantes comme le nombre de neurones de la couche cachée, le nombre d'inputs que nous passons à notre réseau avant que celui-ci calcule le gradient de l'erreur et qu'il adapte ses poids synaptiques et enfin le nombre de fois où nous passons l'ensemble du jeu d'entraînement au réseau. Par manque de temps et à cause d'un temps de préparation des données et d'entraînement important, je n'ai pas pu tester de modifier ces paramètres dans le cadre de mon travail de Bachelor.

```
122 ## Build the model
123 # Model variables
124 n_hidden = 30
125 batch_size = 64
126 n_epoch = 1
```

(Gülen, 2018)

Nous allons maintenant importer les objets dont nous aurons besoin de la librairie Keras citée à la section « 5.1 TensorFlow ». A présent, nous pouvons définir la couche d'inputs qui aura une dimension de « 50 * 300 » pour chaque phrase en entrée. Ici nous définissons une dimension de « None * 300 » pour permettre à la couche de s'adapter à la taille de chacune des phrases et pour permettre à de futurs tests, avec une longueur de phrase maximum plus petite ou plus grande, d'être effectués en changeant un minimum de paramètres. Nous définissons le type des inputs à « float32 » qui correspond à des nombres réels et nous nommons les inputs du côté

Rapport-Gratuit.com

gauche de notre Siamese network « input_1 » et ceux du côté gauche « input_2 ». Ici les inputs n'ont que deux dimensions, mais lors du lancement de l'entraînement, Tensorflow va créer un Tensor en 3 dimensions en ajoutant comme troisième dimension le nombre de paires du jeu d'entraînement. Nous définissons ensuite à la ligne 137 la couche cachée grâce à la méthode « LSTM » de la librairie Keras. Celle-ci aura comme fonction d'activation « relu », se nommera « lstm_1_2 » et recevra en entrée des phrases de dimensions 50 * 300. Les lignes 139 et 140 nous permettent de stocker les vecteurs de sortie de la couche cachée pour pouvoir ensuite calculer la distance entre ceux-ci.

```
128 ## Importing required keras libraries
129 from keras.models import Model
130 from keras.layers import LSTM, Input, Merge
131 import keras.backend as K
132 # The visible layer
133 left_input = Input(shape=(None,300), dtype='float32', name = 'input_1')
134 right_input = Input(shape=(None,300), dtype='float32', name = 'input_2')
135
136 # Since this is a siamese network, both sides share the same LSTM
137 shared_lstm = LSTM(n_hidden, activation = 'relu', name = 'lstm_1_2', input_shape=X_train['left'].shape)
138
139 left_output = shared_lstm(left_input)
140 right_output = shared_lstm(right_input)
```

(Gülen, 2018)

Ici nous définissons une fonction qui prendra deux vecteurs en paramètre et retournera la distance Manhattan entre ceux-ci. Nous devions normalement effectuer la distance Cosinus, mais après quelques tests, celle-ci fournissaient quasiment toujours le même résultat et j'ai donc dû prendre la décision d'essayer avec un autre type de distance qui a été la distance Manhattan. La formule de celle-ci se présente ainsi :

$$d=\sum_{i=1}^{n} |x_i - y_i|$$

Cette distance consiste simplement à effectuer la somme de la soustraction de chaque indice du vecteur x avec chaque indice du vecteur y. Or dans notre cas, certaines valeurs des vecteurs x et y peuvent être négative et nous voulons prédire finalement une valeur entre 0 et 1. Nous allons donc simplement mettre au carré le résultat à l'aide de la fonction « exp » de Keras. Le reste de la formule est la somme de la différence absolue des deux vecteurs et est effectuée à l'aide des fonctions « abs » pour absolue et « sum » pour somme de Keras.

```
143 def exponent_neg_manhattan_distance(left, right):
144     '''
145     Purpose : Helper function for the similarity estimate of the LSTMs outputs
146     Inputs : Two n-dimensional vectors
147     Output : Manhattan distance between the input vectors
148     '''
149     return K.exp(-K.sum(K.abs(left-right), axis=1, keepdims=True))
```

J'aurais pu utiliser la distance Euclidienne plutôt que la distance Manhattan dans le cadre de ce travail de Bachelor, mais c'est de nouveau par manque de temps et par la simplicité à trouver cette formule que je l'ai choisie.

Pour pouvoir utiliser cette formule avec les sorties du réseau de neurones, voici le code que nous devons effectuer :

```
151 # Calculates the distance as defined by the MaLSTM model
152 malstm_distance = Merge(mode=lambda x: exponent_neg_manhattan_distance(x[0], x[1]),
153                         output_shape=lambda x: (x[0][0], 1))([left_output, right_output])
```

(Gülen, 2018)

Pour expliquer ce que fait la ligne 152 et 153, je vais les décomposer en deux parties. Tout d'abord nous allons utiliser le mot clé lambda qui nous permet d'utiliser une fonction anonyme en python. Cela nous évite de devoir déclarer une fonction avec un nom et de devoir ensuite l'appeler :

```
lambda x: exponent_neg_manhattan_distance(x[0], x[1])
```

(Gülen, 2018)

Donc nous faisons appel au mot clé lambda avec « x » comme argument qui est un vecteur contenant deux paramètres (nos deux vecteurs), x [0] et x [1]. Lambda doit toujours être appelée avec au moins un argument et une expression (W3School.com, 2018) et ici l'expression est notre formule calculant la distance de Manhattan.

```
Merge(mode=lambda x: exponent_neg_manhattan_distance(x[0], x[1]),
      output_shape=lambda x: (x[0][0], 1))([left_output, right_output])
```

(Gülen, 2018)

Nous avons besoin d'une couche de Merge fournie par la librairie Keras dans le but de transformer le résultat du calcul de la distance de Manathan en un Tensor pour l'étape suivante. Le paramètre « output_shape » de la couche de Merge correspond à la dimension de sortie de notre Tensor. Enfin les paramètres « left_output » et « right_output » correspondent aux deux vecteurs de sortie de notre Siamese network et sont injectés dans l'argument « x » lors de l'entraînement du réseau.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

Nous pouvons maintenant utiliser la couche appelée Model de la librairie Keras qui prend en paramètres les inputs et les outputs. Ici nous avons deux côtés dans notre Siamese network, il y a donc un vecteur contenant les entrées de gauche et les entrées de droite, ainsi que le Tensor de sortie qui sera le résultat de la distance de Manhattan qui seront injectées lors du lancement de l'entraînement. Pour revenir sur le paragraphe précédent, c'est à cette étape que la couche Model prend obligatoirement un Tensor en sortie du réseau, car il ne connait que ce type de structure.

```
155 # Combine all of the above in a Model
156 model = Model([left_input, right_input], [malstm_distance])
```

(Gülen, 2018)

Enfin, nous devons compiler le modèle pour qu'il puisse initialiser toutes ses variables et devons y définir une fonction d'optimisation et une fonction de coût. La fonction de coût sera appliquée sur les prédictions effectuées toutes les 64 paires de phrases, qui est notre « batch_size » défini quelques paragraphes plus haut, et va calculer la moyenne de l'erreur au carré (différence entre la valeur à prédire et celle prédite) pour ensuite adapter les poids dans le but de réduire cette erreur. Quant à la fonction d'optimisation, il s'agit d'une fonction qui va tenter d'optimiser la descente du gradient de l'erreur qui a été abordé dans la section « 2.2 Le Perceptron multi couches » et nous utilisons « Adam » comme algorithme car il va automatiquement adapter le « Learning rate » (c.f section « 5.2.1.2 Hyperparamètres ») durant l'entraînement comparé à l'algorithme standard appelé « Sochastic gradient descent ». Je ne rentrerais malheureusement pas plus dans les détails de fonctionnement de ces deux algorithmes dans le cadre de ce travail de Bachelor.

Une fois compilé, nous allons déclarer un répertoire de stockage pour y mettre un fichier pouvant être exécuter avec l'outil Tensorboard, cité à la section « 5.1.2.1 Scope », pour pouvoir visualiser des histogrammes sur l'évolution de la précision et de la fonction de coût à travers les cycles d'entraînement. De plus, nous allons stocker uniquement le meilleur cycle d'entraînement sur la base du résultat de la fonction de coût le plus bas sur le jeu de validation. Nous allons enregistrer uniquement les poids du réseau du fait que le modèle en entier pèse presque 1 GB.

```
161 model.compile(loss='mean_squared_error', optimizer= 'adam', metrics=['accuracy'])
162
163 from keras.callbacks import TensorBoard
164 tensor_board = TensorBoard(log_dir='/tmp/Keras_logs/Graph', histogram_freq=0)
165 mcp = ModelCheckpoint('/tmp/Keras_logs/Best_MaLstmWeight.h5', monitor='val_loss', verbose=0 ,
166                       save_best_only=True, save_weights_only=True, mode='min', period=1)
```

(Gülen, 2018)

Nous pouvons maintenant lancer l'entraînement en lui passant les questions en entrée, en déterminant un nombre « d'epochs » assez important pour éviter de sous entraîner notre réseau et nous attribuons au paramètre « callbacks » l'objet « ModelCheckpoint » et « Tensorboard » crée juste au-dessus.

```
168 # Start training
169 model.fit([X_train_left, X_train_right], y_train, batch_size=batch_size, epochs=50,
170                           validation_data=([X_val_left, X_val_right], y_val), callbacks=[mcp, tensor_board])
```

(Gülen, 2018)

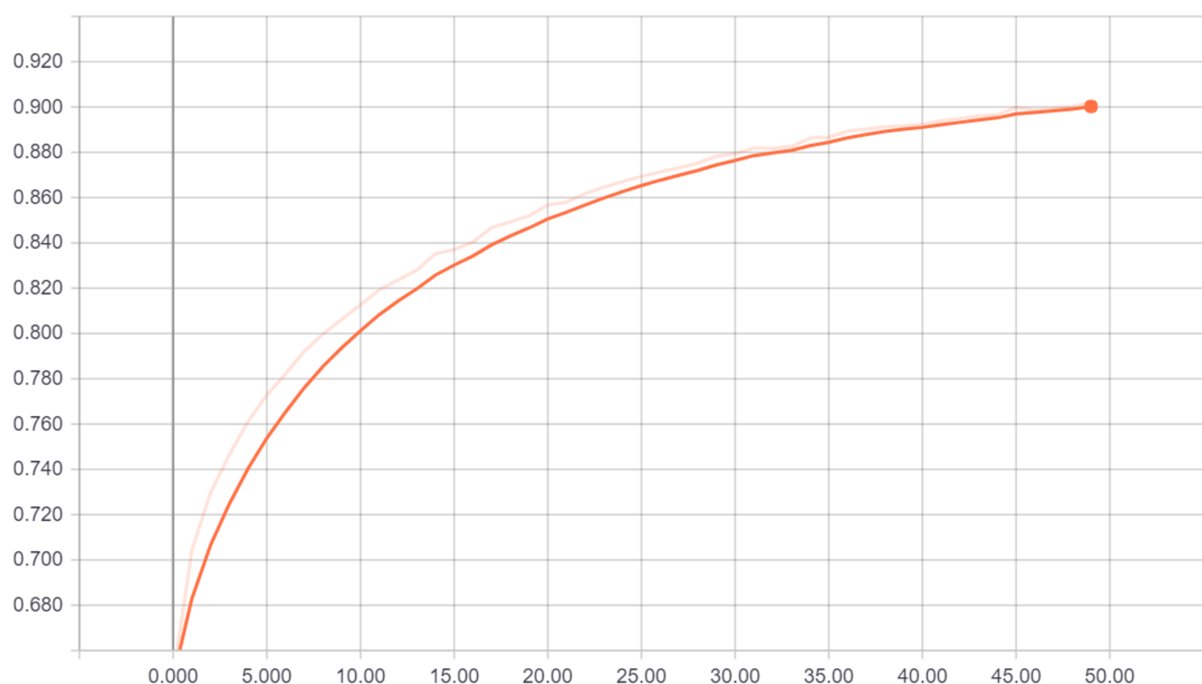
6.4 Résultats

Nous voici dans cette rubrique pour parler des résultats de l'entraînement du modèle après avoir effectué 50 « epochs ».

Pour une visualisation plus intuitive de l'évolution de la précision et du résultat de la fonction de coût du modèle à travers les « epochs », nous allons utiliser les histogrammes générés automatiquement par l'outil Tensorboard. Noté que l'axe des abscisses représente le nombre « d'epoch ».

On peut voir sur la Figure 41, qui a été effectuée sur les résultats du jeu d'entraînement avec notamment la précision de ses prédictions sur l'axe des ordonnées, que la précision est en constante augmentation au fil des « epochs » jusqu'à atteindre environ 90%. Cela signifie que plus on fait de cycles d'entraînement, plus le réseau émet des prédictions juste sur le jeu d'entraînement.

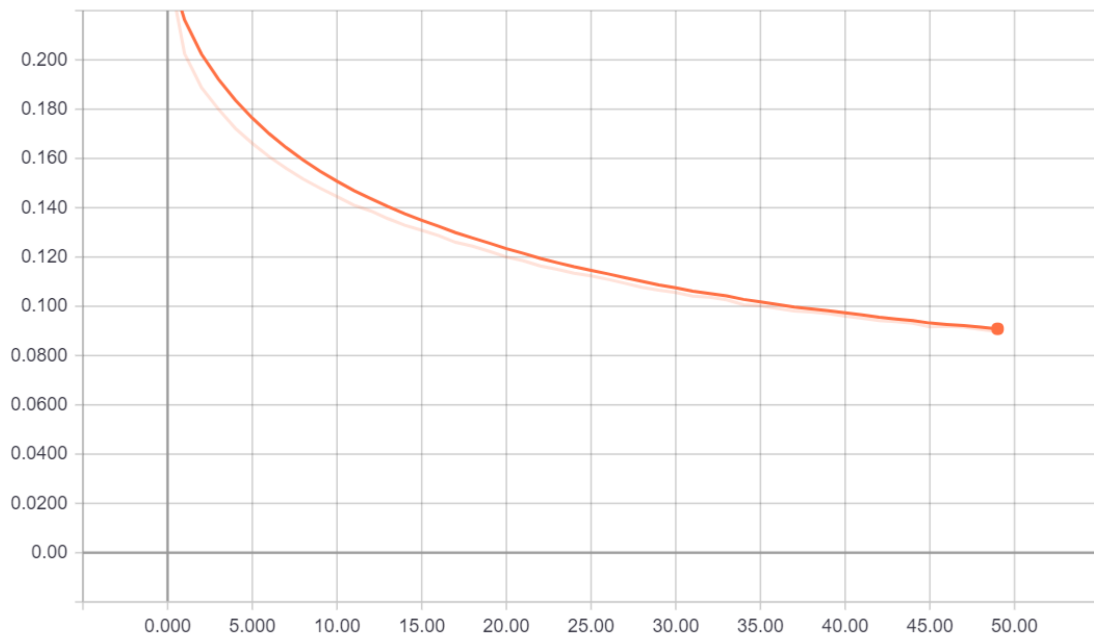
Figure 41 Graphique de la précision du modèle sur le jeu d'entraînement



(Gülen, 2018)

Voyons désormais le résultat de la fonction de coût et son évolution à travers les epochs :

Figure 42 Graphique du résultat de la fonction de coût sur le jeu d'entraînement

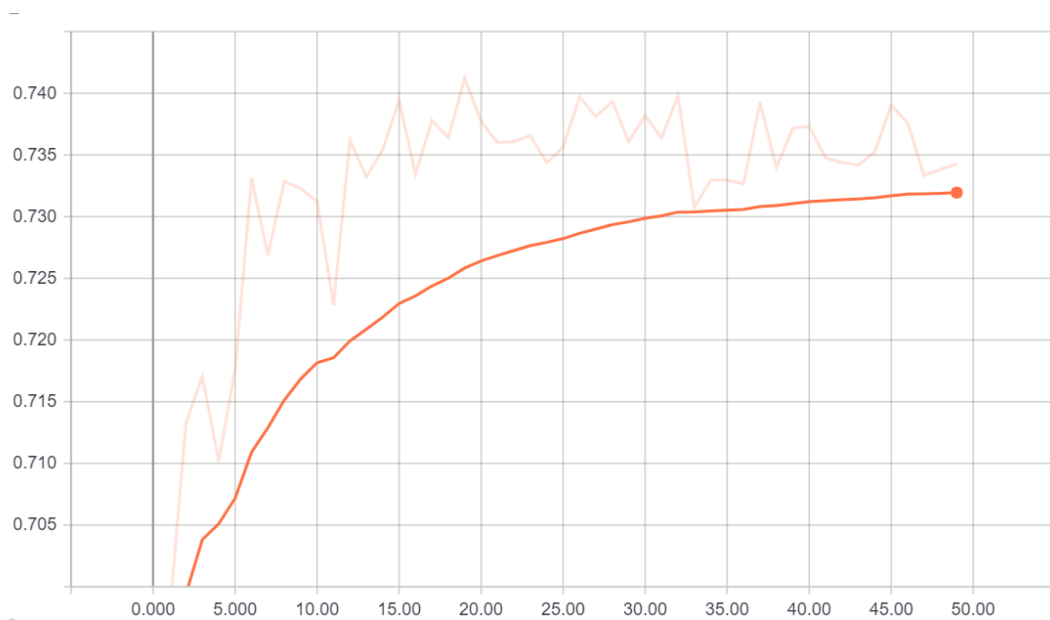


(Gülen, 2018)

Il semble logique que la fonction de coût diminue elle aussi à travers les epochs du fait que la précision du modèle augmente également. C'est en effet en diminuant le résultat de la fonction de coût que le réseau est capable de diminuer son taux d'erreur et de faire de meilleures prédictions.

En outre, voici un autre graphique qui lui a été effectué sur le jeu de validation et va nous permettre de déterminer si les graphiques précédents représentent une réelle amélioration du réseau à travers les cycles d'entraînement où s'il commence à surentraîner.

Figure 43 Graphique de la précision sur le jeu de validation

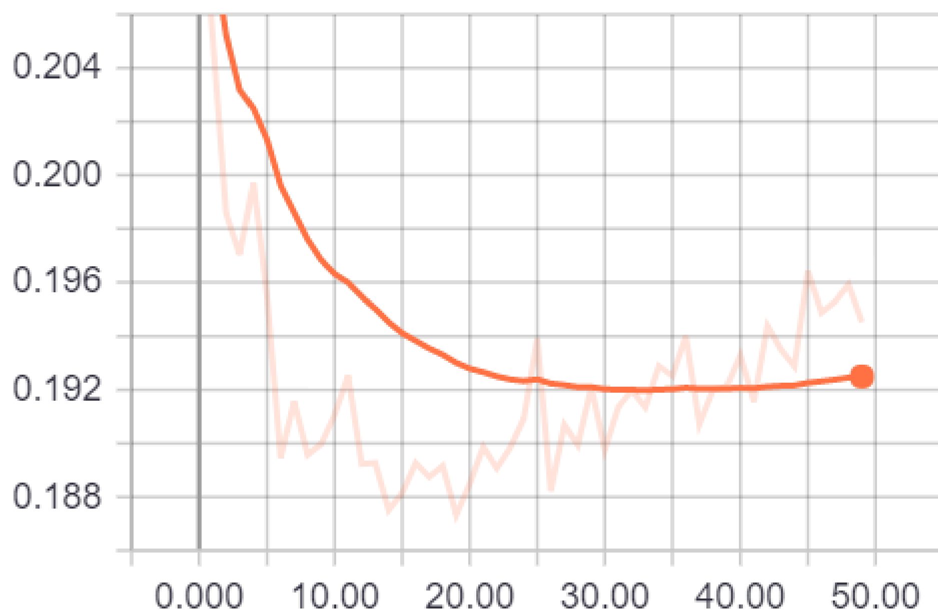


(Gülen, 2018)

Sur la Figure 43 on peut observer que la précision des prédictions s'améliore progressivement jusqu'à atteindre environ 74 %. Contrairement à la précision sur le jeu d'entraînement qui était montée jusqu'à 90%. On peut déjà avoir un indice quant au surentraînement probable de notre réseau mais à ce stade nous ne pouvons pas dire à partir de quel cycle d'entraînement ceci commence.

Voyons donc le résultat de la fonction de coût sur le jeu de validation pour pouvoir enfin en tirer des conclusions :

Figure 44 Graphique du résultat de la fonction de coût sur le jeu de validation



(Gülen, 2018)

Sur ce graphique le même phénomène que sur la Figure 43 se produit. En effet, on peut voir la courbe qui voit sa pente diminuer petit à petit jusqu'à stagner et même de remonter un peu sur la fin. Comme sur le jeu d'entraînement, le résultat de la fonction de coût et le taux de précision sont étroitement liés et cette corrélation entre l'évolution des deux courbes des figures 43 et 44 est tout à fait normal.

Enfin, au vu des résultats et plus particulièrement sur le jeu de validation qui présente des phrases sur lesquelles le réseau ne s'est pas entraîné, nous pouvons considérer qu'un nombre « d'epochs » d'environ 30 suffirait à arriver au minimum de la fonction de coût. Aussi nous prendrons la prendrons le taux de précision d'environ 74% de « l'epoch » 30 comme base de référence puisqu'à ce niveau le réseau n'était pas encore surentraîné et qu'à ce niveau on obtient la fonction de coût la plus basse.

7. Test du modèle construit

Nous voici dans la rubrique où nous allons tester notre modèle qui a précédemment été entrainé. Pour ce faire, et comme j'ai décidé de ne pas stocker l'entièreté du modèle mais uniquement ses poids, nous devons déclarer à nouveau la structure de celui-ci et ensuite charger les poids pré-entrainé.

Comme à la section « 6.2API Embeddings », nous allons utiliser la librairie flask pour pouvoir requêter ce service de test en temps réel en lui fournissant simplement des phrases à analyser.

Comme pour l'API d'embeddings, nous devons initialiser notre serveur de test :

```
13 app = flask.Flask(__name__)
14 CORS(app, resources={r"/*": {"origins": "*"}})
15 app.debug = True
```

(Gülen, 2018)

Nous allons ensuite définir une méthode qui sera uniquement exécutée lors du lancement du serveur de test et qui permettra de charger le modèle dans la mémoire RAM. Pour éviter certains problèmes d'allocation mémoire lors de la mise en service du serveur et des appels de celui-ci, nous devons définir la quantité de mémoire que Tensorflow aura à disposition pour faire ses calculs avec ma carte graphique.

```python
21 def load_my_model():
22
23     #This lines set a parameter to avoid : "CUBLAS_STATUS_ALLOC_FAILED"
24     gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.5, allow_growth=True)
25     sess = tf.Session(config=tf.ConfigProto(gpu_options=gpu_options))
26     K.set_session(sess)
```

(Gülen, 2018)

En effet, si cette étape n'est pas effectuée avec ma carte graphique qui ne contient que 3Gb de mémoire vive, Tensorflow essaiera par défaut d'accéder à l'entièreté de cette mémoire et ne réussira pas à le faire si une partie de celle-ci n'est pas disponible. J'ai donc affecté le paramètre « per_process_gpu_memory_fraction » à 0.5 ce qui équivaut à 50% de mise à disposition de la mémoire de ma carte graphique pour Tensorflow.

Il faut à présent définir la structure du modèle comme elle l'a été lors de l'entraînement :

```python
30     n_hidden = 30
31
32     # The visible layer
33     left_input = Input(shape=(None,300), dtype='float', name = 'input_1')
34     right_input = Input(shape=(None,300), dtype='float', name = 'input_2')
35
36
37     # Since this is a siamese network, both sides share the same LSTM
38     shared_lstm = LSTM(n_hidden, activation = 'relu', name = 'lstm_1_2')
39
40     left_output = shared_lstm(left_input)
41     right_output = shared_lstm(right_input)
42
43     # Calculates the distance as defined by the MaLSTM model
44     malstm_distance = Merge(mode=lambda x: exponent_neg_manhattan_distance(x[0], x[1]),
45                             output_shape=lambda x: (x[0][0], 1))([left_output, right_output])
46
47     global model
48     # Combine all of the above in a Model
49     model = Model([left_input, right_input], [malstm_distance])
50
51     # Load saved weights and then compile to make it ready for further training or predictions
52     model.load_weights("model30_relu_epoch_50.h5", by_name = True)
53     model.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])
```

(Gülen, 2018)

A la ligne 47 nous définissons la variable « model » comme étant global pour pouvoir la réutiliser dans la méthode suivante. La seule différence ici avec le modèle d'entraînement réside dans la ligne 52 où on va charger les poids précédemment enregistrés pour que le réseau soit prêt à faire des prédictions avec la meilleure précision obtenue.

La dernière étape du chargement du modèle est contenue dans les deux lignes suivantes qui permettent de déclarer une variable « graph » comme global et d'y charger le graphe de flux de données du modèle que l'on vient de compiler. Celle-ci nous servira également dans la dernière méthode.

```
54      global graph
55      graph = tf.get_default_graph()
```

(Gülen, 2018)

La seconde méthode sert ici à préparer les inputs qui seront insérés dans le réseau puisque nous aimerions éviter à l'utilisateur de le faire lui-même et de n'avoir qu'à envoyer ses phrases tel quel et d'en récupérer la prédiction. Celle-ci s'attend à recevoir en paramètres deux strings représentant les deux phrases à préparer. Ceux-ci ont la même forme que ce que renvoi notre API expliquée à la section « 6.2 API Embeddings ». L'application qui sera donc utilisée par l'utilisateur devra se charger, avant d'envoyer les phrases à notre script python de test, d'envoyer les deux phrases à notre API pour ensuite envoyer ces résultats à la méthode ci-dessous. L'interface graphique qui servira à l'utilisateur ne sera pas présentée dans cette thèse, mais sera présentée lors de la soutenance, car il n'entre pas dans le cadre de ce travail de recherche mais plutôt dans l'aboutissement de l'objectif fixé au départ.

```
57 def prepare_inputs (input_a, input_b) :
58     # We clean the string in order to keep just the values
59     input_a = input_a.replace("[ ", "")
60     input_a = input_a.replace("[", "")
61     input_a= input_a.replace("]","")
62     #We split the string by the "#" character and get an array of string and each index is the string embedding of the
63     #original word in the tense
64     input_a_array = np.array(input_a.split(' # '))
65     #The last index is an empty value so we do not keep it
66     input_a_array = input_a_array[0:len(input_a_array)-1]
67     #As we cannot transform the orginal array of string in array of sequence, we create a new np array containing
68     #only 0 values, with the same length of the orginial np array and which contains at each index a 300d array which
69     #is the embedding representation for a word
70     input_a_prepared = np.zeros((30,300))
71     for i in range(len(input_a_array)) :
72         input_a_prepared[i] = input_a_array[i].split()
```

(Gülen, 2018)

Les traitements ici sont exactement les mêmes que dans la section « 6.2 API Embeddings ». Les lignes 70-71-72 permettent de créer un vecteur en deux dimensions qui représentent ici la première phrase reçue en paramètres et qui a donc 50 vecteurs, chacun d'une longueur de 300. Comme dans la phase d'entraînement, si un mot contenu dans la phrase originale n'est pas trouvé dans le fichier GloVe cité dans la section « 6.2API Embeddings », il sera représenté par un vecteur de 300 zéros et donc ignorés par notre réseau.

Ce traitement est effectué également pour la deuxième phrase comme suit :

```
74      # Same traitment for the second input
75      input_b = input_b.replace("[ ", "")
76      input_b = input_b.replace("[", "")
77      input_b = input_b.replace("]","")
78      input_b_array = np.array(input_b.split(' # '))
79      input_b_array = input_b_array[0:len(input_b_array)-1]
80      input_b_prepared = np.zeros((30,300))
81      for i in range(len(input_b_array)) :
82          input_b_prepared[i] = input_b_array[i].split()
```

(Gülen, 2018)

Avant de retourner les deux phrases maintenant représentées par des vecteurs de nombre réels, nous devons redimensionner ceux-ci pour qu'ils soient en trois dimensions et soient acceptés en entrée de notre Siamese network, qui n'accepte que des Tensor en 3 dimensions.

Pour ce faire, nous allons utiliser la méthode « reshape » directement disponible sur nos tableaux qui sont des « numpy array » et ont donc été créé grâce à la librairie numpy. On rajoute simplement le premier axe qui représente le nombre de phrases que nous passons à notre réseau et qui est donc égal à « 1 » pour chacune des phrases.

```
84      input_a_prepared = input_a_prepared.reshape(1,len(input_a_prepared),300)
85      input_b_prepared = input_b_prepared.reshape(1,len(input_b_prepared),300)
86      return input_a_prepared, input_b_prepared
```

Maintenant que nous avons une méthode pour charger le modèle et une méthode pour préparer les entrées, il ne nous manque plus qu'une méthode que l'utilisateur va pouvoir appeler depuis l'extérieur et qui va s'occuper d'appeler la méthode « prepare_inputs » précédemment définie pour ensuite donner les entrées au model, récupérer la prédiction et la retourner à l'utilisateur.

Pour ce faire, nous pouvons déjà définir la route – une route est une partie de l'URL que l'utilisateur entre en plus de l'adresse du serveur et qui lui permet d'accéder au service voulu - sera la suivante :

```
88 @app.route("/predict", methods=["POST"])
89 def predict():
```

L'utilisateur y accédera donc en effectuant une requête HTTP de type Post en appelant la route « http://127.0.0.1/predict ». La méthode créera l'objet appelé aussi « dictionnaire » en python qui contiendra plusieurs attributs à retourner. Le premier s'appelle « success » et indique à l'utilisateur si tout s'est passé comme prévu et sa valeur par défaut est « False ». Il transforme ensuite le contenu de la requête envoyée

Rapport-Gratuit.com

par l'utilisateur en objet JSON pour pouvoir accéder aux deux attributs attendus qui sont « tense_1 » et « tense_2 ». On transforme ensuite les deux phrases à l'aide de la méthode « prepare_inputs » précédemment expliquée. Puis on utilise la variable globale « graph » définie précédemment et qui contient le graph de flux de données de notre modèle de Siamese network. Etant donné que Keras rencontre un problème lors de l'exécution de la méthode « predict » lors de la phase de test, nous devons utiliser le graph précédemment déclaré et qui représente le modèle déjà chargé.

Le réseau nous retournant une valeur en 0 et 1, il nous suffit de faire un test pour savoir si elle est plus proche de 0 ou de 1 et de retourner l'attribut « prediction » qui prendra la valeur : « Ces phrases sont similaires » si la valeur est plus grande que 0.5 et « Ces phrases ne sont pas similaires » si la valeur en est inférieur.

On retourne également la valeur prédite à l'utilisateur pour qu'il puisse se rendre compte à quelle niveau ces phrases sont similaires ou non selon notre réseau.

```
92      data = {"success": False}
93      # ensure an image was properly uploaded to our endpoint
94      json_Object = flask.request.get_json()
95      # preprocess the image and prepare it for classification
96      input_left_prepared, input_right_prepared = prepare_inputs(json_Object['tense_1'], json_Object['tense_2'])
97      # classify the input image and then initialize the list
98      # of predictions to return to the client
99      with graph.as_default():
100         ## Making predictions on validation set
101         preds = model.predict([input_left_prepared, input_right_prepared])
102         if preds[0] < 0.5 :
103             data["prediction"] = "Ces phrases ne sont pas similaires"
104         else :
105             data["prediction"] = "Ces phrases sont similaires"
106
107         data["value_predicted"] = float(preds[0])
108         # indicate that the request was a success
109         data["success"] = True
110
111     # return the data dictionary as a JSON response
112     return flask.jsonify(data)
```

(Gülen, 2018)

Et enfin pour que le serveur se lance lorsque le script est exécuté, nous devons appeler la méthode « load_my_model » expliquée plus haut et si tout c'est bien passé, on lance le serveur avec la méthode « run ». Celui-ci est lancé sur le port 5002 puisque le port occupé par l'API d'embedding est le 5001, cependant on aurait pu choisir n'importe quel autre port qui n'est pas utilisé.

```
114 print(("* Loading Keras model and Flask starting server..."
115     "please wait until server has fully started"))
116 load_my_model()
117 print("Model loaded correctly, trying to start the server ...")
118 app.run(host='127.0.0.1', port=5002)
```

(Gülen, 2018)

7.1 Résultats

Penchons-nous dans cette rubrique sur les résultats obtenus après que le script python de test ait été prêt à l'emploi.

Dans un premier temps, je voulais faire une phase de test avec plusieurs milliers de phrases provenant du jeu de données que j'ai utilisé pour entraîner mon réseau de neurones. Cependant, je n'ai pas jugé utile de le faire puisque pendant la phase d'entraînement, le réseau utilise le jeu de validation pour tester de lui-même ses performances sur des données sur lesquelles il ne s'est pas entraîné. On peut donc prendre le taux de précision retourné sur le jeu de validation lors de la phase d'entraînement avec la valeur de la fonction de coût la plus faible.

En effet, même si le taux de précision augmente sur le jeu d'entraînement, cela ne veut pas forcément dire que le réseau s'améliore, mais il peut plutôt faire de l'overfitting, comme expliqué à la section « 5.2.1 Bonnes pratiques » et donc apprendre les données. Il faut donc se baser sur le résultat de la fonction de coût, epochs après epochs, sur le jeu de validation.

Au niveau des résultats, nous avons donc pu voir que notre réseau arrive à un taux de précision de 74%. Cela signifie que notre réseau arrive à prédire correctement plus de 7 paires de phrases sur 10 du jeu de validation qui, je me répète, n'ont pas été insérées dans le jeu d'entraînement et que le réseau n'a donc jamais vues.

Pour continuer la phase de test avec des données plus concrètes, nous pouvons essayer de donner des phrases qui ont une structure et des sujets complètement différents du jeu de données de Quora pour voir à quel point il s'en sort et s'il est véritablement généralisé.

Pour ceci, prenons l'exemple des deux phrases suivantes :

1. « I always loved eating bread in the morning with nutella and especially on Sunday »
2. «I often loved eating some bread with some nutella and especially every Sunday morning.»

Et notre réseau a réussi à trouver un taux de similarité de l'ordre de 60.16%. Ce qui veut dire qu'il les considère comme étant sémantiquement similaires. On peut donc voir que bien que la structure de la phrase ne soit pas mise sous forme interrogative et ne ressemble pas aux phrases du jeu d'entraînement, le réseau arriver tout de même à détecter leur similarité. Cependant, il pourrait s'agir d'un cas exceptionnel et il faudrait

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

donc pouvoir avoir un deuxième jeu de données avec des phrases complètement différentes du jeu d'entraînement pour pouvoir s'assurer que le réseau est fiable. Ceci n'étant pas à ma disposition, je n'entrerai pas plus dans les détails à ce niveau-là dans mon travail de Bachelor.

Maintenant nous pouvons effectuer aussi un test sur deux phrases faisant parties du jeu de données pour voir si le réseau est capable de faire une prédiction satisfaisante. Ceci nous permettra de comparer avec le résultat précédent et de pouvoir en tirer des conclusions dans la rubrique suivante. Voici les deux phrases que nous avons sélectionnées :

1. « How is the new Harry Potter book 'Harry Potter and the Cursed Child' ? »
2. « How bad is the new book by J.K Rowling ? »

Ces deux phrases sont annotées comme étant similaires dans le jeu de données de Quora et notre réseau de neurones a réussi à les prédire similaires aussi avec un taux de 52.81% de similarité.

8. Synthèse et perspectives

Notre Siamese Network est fonctionnel et obtient des résultats déjà satisfaisants. Cependant, quelque chose l'empêche d'apprendre plus et d'augmenter son taux de précision, particulièrement sur le jeu de validation. Je pense que plusieurs options d'amélioration pourraient être envisagées pour tenter de le rendre encore meilleure.

Premièrement, il faudrait avoir une machine plus performante pour pouvoir entraîner le réseau sur tout le jeu de données et lui donner donc plus d'exemples. De plus, il faudrait avoir un jeu de données qui soit plus varié que celui de Quora. En effet, toutes les questions ont une structure très similaire et ceci ne permet pas de généraliser de manière convenable notre réseau de neurones artificiels.

Deuxièmement, il faudrait pouvoir nettoyer un maximum le jeu de données avant d'essayer d'obtenir les « word embeddings » des mots qui y sont contenus. C'est-à-dire de supprimer tous les caractères que ce soit des «, », des «. », des « ;», etc. Par ailleurs, il serait intéressant de réussir à entraîner les « Word embeddings » des mots qui n'ont pas de représentation et qui sont donc ignorés avec un vecteur de zéros. On pourrait ainsi s'assurer d'avoir une représentation pour chacun des mots.

Enfin, il faudrait essayer de modifier la fonction d'activation utilisée avec d'autres, d'augmenter ou de diminuer le nombre de neurones de la couche cachée et d'utiliser

d’autres calculs de distance que la distance de Manhattan sur la dernière couche, afin de voir si notre modèle peut faire de meilleures prédictions.

9. Conclusion

Les réseaux de neurones ne sont pas, en toute généralité, si mistiques que ça en fin de compte. En effet, j'avais souvent entendu parler de réseaux de neurones artificiels et leur capacité à prendre des décisions tous seuls une fois qu'ils étaient entraînés. De plus, on me disait souvent : « Si ça marche, tant mieux. Mais on ne sait pas vraiment pourquoi ». Mais après toutes ces recherches et mes tentatives de créer un modèle le plus performant possible, je commence à mieux saisir chaque partie d'un réseau de neurones. Dans le fond, ce n'est que des mathématiques. Cependant, il n'en est pas moins difficile de pouvoir utiliser et comprendre un assemblage de plusieurs algorithmes mathématiques complexes que représentent les réseaux de neurones artificiels.

Je ne peux pas dire que l'objectif initial, qui était de permettre une automatisation de la correction des travaux des étudiants, a été atteint. Néanmoins, je suis content d'en être arrivé jusque-là et les résultats que j'ai pu obtenir me confortent dans l'idée qu'avec plus de temps à ma disposition et plus de puissance de calcul, je pourrais peaufiner et améliorer le modèle finalement construit.

En outre, je pense que mon travail ne s'inscrit pas seulement dans la recherche d'une solution d'automatisation d'un processus de correction. Il soulève, en tout cas chez moi, des questions par rapport au rôle que joue le « Natural Language Processing » dans le domaine du Machine Learning.

Selon moi, c'est la maîtrise de cette discipline qui manque aux différents objets connectés comme les synthétiseurs vocaux et d'autres. Si la « machine » était capable de réellement comprendre le sens de ce qu'on aimerait lui dire, sans que ceci soit pré-enregistré, je pense que ceci serait le début d'une réelle intelligence artificielle.

Enfin, j'espère que ce travail de recherche pourra servir à quelqu'un qui, comme moi, s'intéresse à l'analyse de la langue naturelle au moyen de réseau de neurones artificiels.

10. Bibliographie

AMAZON, 2018. Splitting the Data into Training and Evaluation Data - Amazon Machine Learning. In : [en ligne]. 2018. [Consulté le 17 juillet 2018]. Disponible à l'adresse : https://docs.aws.amazon.com/machine-learning/latest/dg/splitting-the-data-into-training-and-evaluation-data.html.

ARGODEV, 2017. 인공신경망(ANN)과 Word2Vec. In : *ARGONET* [en ligne]. 7 mai 2017. [Consulté le 5 mai 2018]. Disponible à l'adresse : https://argonet.co.kr/%ec%9d%b8%ea%b3%b5%ec%8b%a0%ea%b2%bd%eb%a7%9dann%ea%b3%bc-word2vec/.

BALDI, Pierre et CHAUVIN, Yves, 1993. *Neural Networks for Fingerprint Recognition*. No. 3. S.l. : Terrence Sejnowski. ISBN 0899-7667.

BHEEMAIAH, Kariappa, ESPOSITO, Mark et TSE, Terence, 2017. Deep learning, des réseaux de neurones pour traiter l'information. In : *The Conversation* [en ligne]. 2 mai 2017. [Consulté le 3 mai 2018]. Disponible à l'adresse : http://theconversation.com/deep-learning-des-reseaux-de-neurones-pour-traiter-linformation-76055.

BOJANOWSKI, Piotr, GRAVE, Edouard, JOULIN, Armand et MIKOLOV, Tomas, 2016. fastText. In : *Facebook Research* [en ligne]. 2016. [Consulté le 26 juin 2018]. Disponible à l'adresse : https://research.fb.com/fasttext.

BRITZ, Denny, 2015. Understanding Convolutional Neural Networks for NLP. In : *WildML* [en ligne]. 7 novembre 2015. [Consulté le 6 mai 2018]. Disponible à l'adresse : http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/.

BROWNLEE, Jason, 2016a. Dropout Regularization in Deep Learning Models With Keras. In : *Machine Learning Mastery* [en ligne]. 20 juin 2016. [Consulté le 17 juillet 2018]. Disponible à l'adresse : https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/.

BROWNLEE, Jason, 2016b. Overfitting and Underfitting With Machine Learning Algorithms. In : *Machine Learning Mastery* [en ligne]. 21 mars 2016. [Consulté le 17 juillet 2018]. Disponible à l'adresse : https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/.

BROWNLEE, Jason, 2017a. A Gentle Introduction to Backpropagation Through Time. In : *Machine Learning Mastery* [en ligne]. 23 juin 2017. [Consulté le 18 juin 2018]. Disponible à l'adresse : https://machinelearningmastery.com/gentle-introduction-backpropagation-time/.

BROWNLEE, Jason, 2017b. How Much Training Data is Required for Machine Learning? In : *Machine Learning Mastery* [en ligne]. 24 juillet 2017. [Consulté le 17 juillet 2018]. Disponible à l'adresse : https://machinelearningmastery.com/much-training-data-required-machine-learning/.

COMMONCRAWL.ORG, 2007. Common Crawl. In : [en ligne]. 2007. [Consulté le 31 juillet 2018]. Disponible à l'adresse : http://commoncrawl.org/.

COUCHOT, Jean-François et GUYEUX, Christophe, 2008. WikiMath » RNN/Apprentissage? In : [en ligne]. 2008. [Consulté le 6 mai 2018]. Disponible à l'adresse : http://cours-info.iut-bm.univ-fcomte.fr/wiki/pmwiki.php/RNN/Apprentissage.

CROUSPEYRE, Charles, 2017. Comment les Réseaux de neurones à convolution fonctionnent. In : *Medium* [en ligne]. 17 juillet 2017. [Consulté le 6 mai 2018]. Disponible à l'adresse : https://medium.com/@CharlesCrouspeyre/comment-les-r%C3%A9seaux-de-neurones-%C3%A0-convolution-fonctionnent-b288519dbcf8.

DATAANALYTICSPOST, 2018. Réseaux de neurones récurrents. In : *Data Analytics Post* [en ligne]. 2018. [Consulté le 6 mai 2018]. Disponible à l'adresse : https://dataanalyticspost.com/Lexique/reseaux-de-neurones-recurrents/.

DEAN, Jeffrey et GHEMAWAT, Sanjay, 2008. MapReduce: simplified data processing on large clusters. In : *Communications of the ACM*. 1 janvier 2008. Vol. 51, n° 1, p. 107. DOI 10.1145/1327452.1327492.

DERNONCOURT, Franck, 2017. machine learning - Is Convolutional Neural Network (CNN) faster than Recurrent Neural Network (RNN)? In : *Cross Validated* [en ligne]. 2017. [Consulté le 6 mai 2018]. Disponible à l'adresse : https://stats.stackexchange.com/questions/262220/is-convolutional-neural-network-cnn-faster-than-recurrent-neural-network-rnn?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa.

DESPOIS, Julien, 2016. Why do we need adaptive learning rates for Deep Learning? - Quora. In : [en ligne]. 2016. [Consulté le 18 juillet 2018]. Disponible à l'adresse : https://www.quora.com/Why-do-we-need-adaptive-learning-rates-for-Deep-Learning.

DOUKKALI, Firdaouss, 2017. Convolutional Neural Networks (CNN, or ConvNets). In : *Firdaouss Doukkali* [en ligne]. 28 septembre 2017. [Consulté le 20 juin 2018]. Disponible à l'adresse : https://medium.com/@phidaouss/convolutional-neural-networks-cnn-or-convnets-d7c688b0a207.

DUC-JACQUET, Patrice, 2017. CNN (Convolution Neural Network): une introduction. In : *Pat's Technical Place (Big Data & Data Science)* [en ligne]. 4 juillet 2017. [Consulté le 20 juin 2018]. Disponible à l'adresse : https://patducjacquet.wordpress.com/2017/07/04/cnn-convolution-neural-network-une-introduction/.

DUPRÉ, Xavier, 2017. Apprentissage d'un réseau de neurones — Machine Learning, Statistiques et Programmation. In : [en ligne]. 2017. [Consulté le 6 mai 2018]. Disponible à l'adresse : http://www.xavierdupre.fr/app/mlstatpy/helpsphinx/c_ml/rn_6_apprentissage.html.

FILY, Quentin, 2015. TF-IDF, occurrence et rareté d'un mot pour le SEO - Quentin Fily. In : [en ligne]. 2 novembre 2015. [Consulté le 25 juin 2018]. Disponible à l'adresse : https://www.quentinfily.fr/tf-idf-pertinence-lexicale/.

GBMB.ORG, 2018. MB to Bytes Conversion Megabytes to Bytes Calculator. In : [en ligne]. 2018. [Consulté le 4 août 2018]. Disponible à l'adresse : https://www.gbmb.org/mb-to-bytes.

GÉRON, Aurélien, 2017. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. S.l. : O'Reilly Media, Inc. ISBN 978-1-4919-6224-4.

HAPNES STRAND, Hâkon, 2016. What is one-hot encoding and when is it used in data science? - Quora. In : [en ligne]. 2016. [Consulté le 25 juin 2018]. Disponible à l'adresse : https://www.quora.com/What-is-one-hot-encoding-and-when-is-it-used-in-data-science.

INTEL.COM, 2017. Hands-On AI Part 5: Select a Deep Learning Framework | Intel® Software. In : [en ligne]. 2017. [Consulté le 11 juillet 2018]. Disponible à l'adresse : https://software.intel.com/en-us/articles/hands-on-ai-part-5-select-a-deep-learning-framework.

JAYANT, Jain, 2016. FastText and Gensim word embeddings | RARE Technologies. In : [en ligne]. 2016. [Consulté le 26 juin 2018]. Disponible à l'adresse : https://rare-technologies.com/fasttext-and-gensim-word-embeddings/.

JAYANT, Jain, 2017. Implementing Poincaré Embeddings | RARE Technologies. In : [en ligne]. 2017. [Consulté le 27 juin 2018]. Disponible à l'adresse : https://rare-technologies.com/implementing-poincare-embeddings/#h2-2.

JSON.ORG, 2018. JSON. In : [en ligne]. 2018. [Consulté le 4 août 2018]. Disponible à l'adresse : https://www.json.org/.

KANG, Eugine, 2017. Long Short-Term Memory (LSTM): Concept. In : *Eugine Kang* [en ligne]. 1 septembre 2017. [Consulté le 19 juin 2018]. Disponible à l'adresse : https://medium.com/@kangeugine/long-short-term-memory-lstm-concept-cb3283934359.

L, +Bastien, 2018. Perceptron – Tout savoir sur le plus vieil algorithme de Machine Learning. In : *LeBigData.fr* [en ligne]. 29 mars 2018. [Consulté le 3 mai 2018]. Disponible à l'adresse : https://www.lebigdata.fr/perceptron-machine-learning.

LECUN, Yann, 2013. Yann LeCun's Home Page. In : [en ligne]. 2013. [Consulté le 27 juin 2018]. Disponible à l'adresse : http://yann.lecun.com/.

MADHU SANJEEVI, Mady, 2018. Chapter 10.1: DeepNLP — LSTM (Long Short Term Memory) Networks with Math. In : *Medium* [en ligne]. 21 janvier 2018. [Consulté le 19 juin 2018]. Disponible à l'adresse : https://medium.com/deep-math-machine-learning-ai/chapter-10-1-deepnlp-lstm-long-short-term-memory-networks-with-math-21477f8e4235.

MCCORMICK, Chris, 2016. Word2Vec Tutorial - The Skip-Gram Model · Chris McCormick. In : [en ligne]. 2016. [Consulté le 25 juin 2018]. Disponible à l'adresse : http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/.

MIKOLOV, Tomas, SUTSKEVER, Ilya, CHEN, Kai, CORRADO, Greg et DEAN, Jeffrey, 2013. Distributed Representations of Words and Phrases and their Compositionality. In : *arXiv:1310.4546 [cs, stat]* [en ligne]. 16 octobre 2013. [Consulté le 25 juin 2018]. Disponible à l'adresse : http://arxiv.org/abs/1310.4546.

MINZ, Jacob, 2016. What deep learning method to use to classify text files? - Quora. In : [en ligne]. 2016. [Consulté le 18 juin 2018]. Disponible à l'adresse : https://www.quora.com/What-deep-learning-method-to-use-to-classify-text-files.

NICHOLAS MCCLURE, 2017. Siamese networks. In : [en ligne]. Data & Analytics. S.l. 2017. [Consulté le 27 juin 2018]. Disponible à l'adresse : https://www.slideshare.net/NicholasMcClure1/siamese-networks.

NICKEL, Maximilian et KIELA, Douwe, 2017. Poincar\'e Embeddings for Learning Hierarchical Representations. In : *arXiv:1705.08039 [cs, stat]* [en ligne]. 22 mai 2017. [Consulté le 26 juin 2018]. Disponible à l'adresse : http://arxiv.org/abs/1705.08039.

NSS, 2017. Word Representations & Text Classification using FastText (Facebook lib). In : *Analytics Vidhya* [en ligne]. 14 juillet 2017. [Consulté le 26 juin 2018]. Disponible à l'adresse : https://www.analyticsvidhya.com/blog/2017/07/word-representations-text-classification-using-fasttext-nlp-facebook/.

OLAH, Christopher, 2015. Understanding LSTM Networks -- colah's blog. In : [en ligne]. 2015. [Consulté le 18 avril 2018]. Disponible à l'adresse : http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

OLEJNIK, Galina, 2017. Word embeddings: exploration, explanation, and exploitation (with code in Python). In : *Towards Data Science* [en ligne]. 3 décembre 2017. [Consulté le 20 juin 2018]. Disponible à l'adresse : https://towardsdatascience.com/word-embeddings-exploration-explanation-and-exploitation-with-code-in-python-5dac99d5d795.

PENNINGTON, Jeffrey, SOCHER, Richard et MANNING, Christopher, 2014. Glove: Global Vectors for Word Representation. In : [en ligne]. S.l. : Association for Computational Linguistics. 2014. p. 1532‑1543. [Consulté le 25 juin 2018]. Disponible à l'adresse : http://aclweb.org/anthology/D14-1162.

PERONE, Christan S., 2013. Machine Learning :: Cosine Similarity for Vector Space Models (Part III) | Terra Incognita. In : [en ligne]. 2013. [Consulté le 27 juin 2018]. Disponible à l'adresse : http://blog.christianperone.com/2013/09/machine-learning-cosine-similarity-for-vector-space-models-part-iii/.

R, Lambert, 2018. Focus : Le Réseau de Neurones Artificiels ou Perceptron Multicouche. In : *Pensée Artificielle* [en ligne]. 11 février 2018. [Consulté le 19 juin 2018]. Disponible à l'adresse : http://penseeartificielle.fr/focus-reseau-neurones-artificiels-perceptron-multicouche/.

ROBBINS, Will, 2017. What is Quora and Why Should You Care? In : *Search Engine Journal* [en ligne]. 10 mai 2017. [Consulté le 18 juillet 2018]. Disponible à l'adresse : https://www.searchenginejournal.com/what-is-quora-and-why-should-you-care/28475/.

ROSENBLATT, F., 1958. *The Perceptron: A Probabilistic Model for Information Storage and Organization*. S.l. : s.n.

ROXOR, Mark, 2016. Word2Vec_FastText_Comparison. In : [en ligne]. 2016. [Consulté le 27 juin 2018]. Disponible à l'adresse : https://markroxor.github.io/gensim/static/notebooks/Word2Vec_FastText_Comparison.html.

RUDER, Sebastian, 2016. An overview of word embeddings and their connection to distributional semantic models. In : *AYLIEN* [en ligne]. 13 octobre 2016. [Consulté le 26 juin 2018]. Disponible à l'adresse : http://blog.aylien.com/overview-word-embeddings-history-word2vec-cbow-glove/.

SCHOONJANS, Frank, 2018. TANH function definition and online calculator. In : *MedCalc* [en ligne]. 2018. [Consulté le 19 juin 2018]. Disponible à l'adresse : https://www.medcalc.org/manual/Tanh_function.php.

SELIVANOV, Dmitriy, 2015. GloVe vs word2vec revisited. · Data Science notes. In : *Data Science notes* [en ligne]. 2015. [Consulté le 26 juin 2018]. Disponible à l'adresse : /post/glove-enwiki/.

SHANKAR, Iyer, NIKHIL, Dandekar et KORNÉL, Csernai, 2017. First Quora Dataset Release: Question Pairs - Data @ Quora - Quora. In : [en ligne]. 2017. [Consulté le 18 juillet 2018]. Disponible à l'adresse : https://data.quora.com/First-Quora-Dataset-Release-Question-Pairs.

SHARMA, Avinash, 2017. Understanding Activation Functions in Neural Networks. In : *Medium* [en ligne]. 30 mars 2017. [Consulté le 19 juin 2018]. Disponible à l'adresse : https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0.

SRIVASTAVA, Nitish, HINTON, Geoffrey, KRIZHEVSKY, Alex, SUTSKEVER, Ilya et SALAKHUTDINOV, Ruslan, 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. In : . 2014. p. 30.

STATISTICA, 2013. Réseaux de Neurones. In : [en ligne]. 2013. [Consulté le 5 mai 2018]. Disponible à l'adresse : http://www.statsoft.fr/concepts-statistiques/reseaux-de-neurones-automatises/reseaux-de-neurones-automatises.htm#.Wu3Rj4iFOM8.

TENSORFLOW.ORG, 2018. Tensors. In : *TensorFlow* [en ligne]. 2018. [Consulté le 11 juillet 2018]. Disponible à l'adresse : https://www.tensorflow.org/guide/tensors.

UNRUH, Amy, 2017. What is the TensorFlow machine intelligence platform? In : *Opensource.com* [en ligne]. 2017. [Consulté le 11 juillet 2018]. Disponible à l'adresse : https://opensource.com/article/17/11/intro-tensorflow.

VARGAS, Esteban, 2017. A Comprehensive Introduction to Word Vector Representations. In : [en ligne]. 2017. [Consulté le 25 juin 2018]. Disponible à l'adresse : https://medium.com/ai-society/jkljlj-7d6e699895c4.

VASUDEV, Rakshith, 2017. What is One Hot Encoding? Why And When do you have to use it? In : *Hacker Noon* [en ligne]. 3 août 2017. [Consulté le 25 juin 2018]. Disponible à l'adresse : https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f.

W3SCHOOL.COM, 2018. Python Lambda. In : [en ligne]. 2018. [Consulté le 4 août 2018]. Disponible à l'adresse : https://www.w3schools.com/python/python_lambda.asp.

WHITE, Nick, 2017. The Problem of Vanishing Gradients. In : *Nick White* [en ligne]. 9 novembre 2017. [Consulté le 6 mai 2018]. Disponible à l'adresse : https://medium.com/@nickikwhite_5051/the-problem-of-vanishing-gradients-330fc874d3e3.

WIKIHOW, 2018. How to Construct a 30 Degrees Angle Using Compass and Straightedge. In : *wikiHow* [en ligne]. 2018. [Consulté le 28 juin 2018]. Disponible à

Rapport-Gratuit.com

l'adresse : https://www.wikihow.com/Construct-a-30-Degrees-Angle-Using-Compass-and-Straightedge.

WIKIPEDIA, 2017. *Neurone formel* [en ligne]. S.l. : s.n. [Consulté le 5 mai 2018]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=Neurone_formel&oldid=143751155.

WIKIPEDIA, 2018a. *Fonction d'activation* [en ligne]. S.l. : s.n. [Consulté le 5 mai 2018]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=Fonction_d%27activation&oldid=147444826.

WIKIPEDIA, 2018b. *Perceptron multicouche* [en ligne]. S.l. : s.n. [Consulté le 5 mai 2018]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=Perceptron_multicouche&oldid=146910799.

WIKIPEDIA, 2018c. *Réseau de neurones artificiels* [en ligne]. S.l. : s.n. [Consulté le 18 juin 2018]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=R%C3%A9seau_de_neurones_artificiels&oldid=147833956.

WIKIPEDIA, 2018d. *Réseau neuronal convolutif* [en ligne]. S.l. : s.n. [Consulté le 6 mai 2018]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=R%C3%A9seau_neuronal_convolutif&oldid=147692696.

WIKIPEDIA, 2018e. *Rétropropagation du gradient* [en ligne]. S.l. : s.n. [Consulté le 5 mai 2018]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=R%C3%A9tropropagation_du_gradient&oldid=145381051.

WINTHROP, Nelson Francis et KUCERA, Henri, 1979. Brown Corpus Manual. In : [en ligne]. 1979. [Consulté le 27 juin 2018]. Disponible à l'adresse : http://clu.uni.no/icame/brown/bcm.html.